Department of Computer Science

Technical Report

Undergraduate Dissertation: Call Graphing in C

Ian Saunders

Technical Report 2007-07 **July 2007**

ISSN 1740-9497

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

CALL GRAPHING IN C

Submitted by Ian Saunders
for the degree of
BSc (Hons) Computer Science
of the University of Bath
2007

Signed:

Call Graphing in C

Submitted by: Ian Saunders

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifcally acknowledged, it is the work of the author.

Signed:

Abstract

This dissertation looks into the various stages of call graph creation. Call graphs provide a structural representation of source code and are often used as a means to understanding source code. A big challenge of call graph extraction, especially so in C, is calling a function indirectly via function pointers. The question is raised of how to solve this problem? Techniques for call graph extraction are implemented in conjunction with existing pointer analysis algorithms. Secondly, visualisation of the call graph are investigated. This work resulted in a new method to identify unused or dead function blocks. Finally a comparison of classic and force-directed graphs combined with novel uses is carried out.

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost the author would like to thank Dr. Marina De Vos. Without her willingness to take on an extra student the author would never have embarked on this dissertation. Useful advice and an always open door have been invaluable assets throughout the year. Thanks are due to Martin Braine for suggesting the original concept of Call Graphing in C and initial interest which excited the author to take on the project. Finally the author would like to thank Professor John ffitch for making compilers an interesting and thought provoking subject.

I am also grateful to all my friends, colleagues and enemies who have made the Bath Computer Science department what it is. Special thanks to Samantha Crawford who read the many drafts of this project and listened to the author wax-lyrical about various inane subjects. Finally thanks to the reader for spending the time to undestand what is, in my opinion, a very interesting problem.

Chapter 1

Introduction

1.1 Overview

It is often said that reading source code is the best way of understanding a given program. Reading the source allows developers to fully understand what a block of code aims to achieve. More importantly it shows the methodology behind the solution. Unfortunately this is easier said than done. With so much information being presented it is often difficult to ascertain what the important parts of the code are. Information such as critical paths, bottlenecks or even dead code are not easily identifiable. As a corollary, it is also easy to get lost in implementation details, forgetting the structural overview. An abundance of programming languages with their own special syntax and structure further compound the problem. A better method for understanding code must be available, hopefully allowing developers to conceptualise an application at a high level presenting the structure succinctly.

One area of research that deals with this problem is that of call graph extraction, extracting the various call chains present in a program. Call graph extraction can then be used to represent the source code in a directed graph known as a Call Graph. Creation of a call graph usually is achieved in two stages. Firstly the source code must be analysed for function calls; resulting in an abstract representation of the code. This abstraction can then be turned into a visual representation of the program.

Unfortunately, the extraction of the call graph is not a trivial process. Languages such as C contain conditional statements, making it impossible to predict the exact application flow. Ignoring these statements reduces the accuracy of our call graph. These languages also provide ways of calling functions other than via direct invocation; Variable types known as

function pointers allow indirect calls. These two problems can lead to imprecise and in the worst case incorrect call graphs. When using these call graphs for debugging this metadata is extremely useful, allowing the developer to infer characteristics of the code.

This dissertation looks into both stages of call graph creation. Methods for call graph extraction are implemented and tested using existing algorithms for pointer analysis. The use of contrasting analysis techniques leads to the discovery of an interesting side effect: discovery of dead function blocks. Various ways of visualising the abstract direct graph are investigated, novel techniques based on dynamic graphs showing caller-caller relationships are also used. This technique may conceivably aid in visualising call graphs and call chains. Finally, a thorough analysis of the various extraction techniques and their associated graphs is carried out.

1.2 Document Guide

This document consists of a number of distinct parts. The structure is based around extracting and then representing a call graph. Firstly an analysis of existing methods and techniques is carried out in the literature review. From here it is possible to begin the process of extracting and creating a call graph. Extraction of call graphs is divided into two chapters. The first of these deals with creating a framework to represent source code in a useful and abstract manner. The second deals with various extraction algorithms. These are based on the code analysis techniques described in the literature review. Now that the call graph has been extracted from code, the next section deals with the visualisation of the call graph. The problems associated with representing calls and functions is discussed, arriving at schema. The use of the schema for each algorithm is discussed and certain standards are arrived at. A discussion of some of the high-level requirements, design process, and implementation details is carried out. Various experiments are devised to test the correctness of the algorithms and to verify if they are getting the expected results. Once these experiments have been carried out an analysis and evaluation of the interesting cases is carried out. An appraisal of work carried out is performed in the final section, finishing on further open research questions relating to call graphing.

Chapter 2

Literature Review

2.1 Historical Context

After an application has been developed it is rarely left as a completed work. Improvements, modifications and bug-fixes are an everyday fact of life. Take for instance a large company such as Microsoft. After developing a piece of software such as Office [Das02] for three or more years, it would be fool-hardy to believe that the software was without problems. To this end, every fortnight fixes are released for people to update their systems. Not only do large software development companies realise this, but it is built into their software development eco-system.

Another factor is the ebb-and-flow of employees and technical staff at these companies. Often a new employee will need to understand and be able to debug a system in a minimal amount of time. It has been seen in empirical testing [HP97, Hin01] that over 50% of the costs of an application over the entire life-cycle can be attributed to maintenance. Various measures are put in place to aid this process. Requirement documents, UML diagrams and other static documents are available to developers [Sun03]. Unfortunately these are often created at design time, and as such do not take into account the actual software implementation. Discussion of abstract problems such as the class hierarchy of the system or work flows are usual for understanding what the program does, and how it should do it, but not the actual implementation of these. One solution, the basis of this dissertation, is that of application call graphs [Ryd79].

2.2 Graphing

Call graphs are a visual way of viewing the implementation of an application. The call graph of an application is a representation of the calling relationships of the various functions in a piece of code [Sha97]. The final output is a directed graph containing nodes and lines between them [ATT07, ATT06].

Take figure 2.2 (page 17) functions are represented by nodes. Thus this graph contains three nodes *a, c, c.* If function *a* calls function *b* an edge is drawn connecting the two nodes. The resulting call graph displays useful data such as:

```
1 int a( void )
2 {
3     b();
4     c();
5 }
```

Figure 2.1: Simple function calls

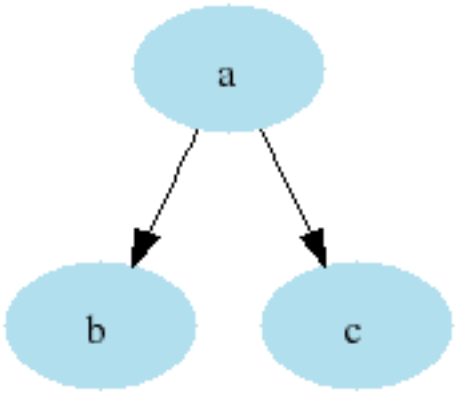


Figure 2.2: Call graph generated from figure 2.1 (page 17)

- Functions contained in an application
- How a function is invoked (called)

When making modifications to a system, it is possible to see what changes will effect which code. Thus a change to function *c* will have no direct effect on *a* or *b*. The overall concept of creating a call graph is relatively simple. Unfortunately, implementation in a language as versatile as C contains many caveats and problems.

2.2.1 Stages in creating a Call Graph

To create a call graph there are two distinct stages. Firstly, the implementation needs to be analysed and the call relationships extracted. The second stage involves the creation of the direct graph from these relationships. The rest of this document concerns itself with these two problems and associated literature.

2.3 Function Calls in C

Function calls can be defined as a method for passing control of an application; in most cases moving from one function to another, then returning to the calling function to complete the initial execution path. The question arises, how can functions be called in a modern programming language, and in particular C? Though this sounds like a trivial question there is more to it than one might first realise. There are two main idioms for calling functions in C either:

- Directly calling a function by name
- Call a function pointer

The process of calling a function by name is relatively trivial. The call is defined at a particular place in the code, and will always have the same semantics with respects to the applications running path. If this were the only method of calling functions in a program it would be possible to read off the call graph straight from the source code.

The other method for calling function via pointers has a great deal more complexity [ACT99, MRR04]. Anyone who has coded any medium to large amount of C would have come across pointers. Pointers are variable types that point to a block of memory on the physical machine and can directly access the value stored by dereferencing these points. As a function has a start point which is defined at a particular place in memory it can be referenced by a pointer. The syntax for calling function calling via the variable is show in the example, figure 2.3 (page 19).

Though at first glance, this method of calling functions may seem convoluted and unnecessary, it is a common paradigm used in software development. It is noted [ACT99] that around 10% of function calls in an average application can be function pointers. This is a significant proportion of calls, and will result in a rather different call graph. To further illustrate the importance of function pointers the following is a short discussion of design patterns associated with function pointers.

```
1  int main()
2  {
3       int x,y;
4
5       int (*sqr)(int) = &square;
6       int (*triple)(int) = &triple;
7
8       x = sqr(65);
9       y = triple(x);
10      x = sqr(y);
11
12      return 0;
13 }
14
15
16 int square(int i)
17 {
18      return i*i;
19 }
20
21 int triple(int i)
22 {
23      return 3*i;
24 }
```

Figure 2.3: Functions called by pointer references

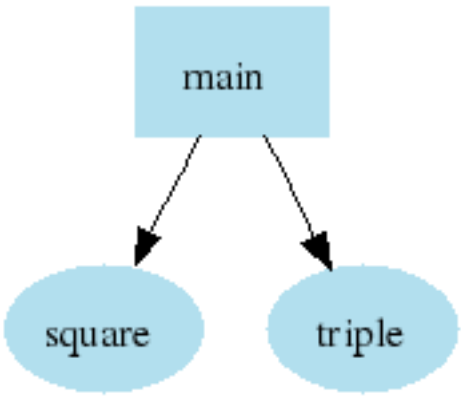


Figure 2.4: Call graph generated from figure 2.3 (page 19)

2.3.1 Function Call Backs

The most obvious use of function pointers is that of creating call backs in applications. When a call is made to a function another functions address can be passed it as part of the formal parameters. This passed in function pointer can then be called whenever it is deemed necessary by the function. Take for instance the situation of wanting to notify the back-end of an application that an event has happened on the user interface, call backs are a good way of implementing this.

2.3.2 Table Dispatch

A popular method which is employed in many languages, table dispatch allows a late binding of variable to function. The idea is as follows; when a specific function is needed a lookup is made based on a key. The lookup happens on a table containing key and function pointer pairs. When the key is found the pointer to the function it is paired with is returned. This pattern is particularly popular and used in a number of commercial applications.

2.3.3 Polymorphic Behaviour

A slightly more complex use of function pointers is the ability to model polymorphic behaviour. This can be implemented by defining a function pointer with a void* return type. This can then allow calls to the same function to behave differently depending on the function parsed in.

2.3.4 Summary

The analysis of both function calls from a normal and pointer perspective is necessary in the C programming language to get a good representation of the application in a call graph. A good understanding of the methods associated with calls is required to extract this information.

2.4 Analysis

There are a large number of applications that can extract a call graph from a given piece of code. Though these often differ quite substantially in implementation, they share a number

of defining characteristics [Ste96].

When analysing a piece of code a number of choices need to be taken on how to understand the code. These decisions determine a number of factors and often there is a trade off between implementations. Often an increase in the accuracy of the call graph results in a greater logical complexity. Other techniques may be quicker but may not be as accurate [And94, Atk04, Ste96, Sun03, EGH94a, Das02, EGH94b].

2.4.1 Flow sensitive vs. Flow in-sensitive

One of the biggest problems is to do with the flow of the application. Given some method the analysis of the flow can be broken up into either flow sensitive or flow in-sensitive.Flow sensitive analysis takes into account the order in which assignments are made. As such when analysing a function each point P_x in the application is treated as a point that has the environment of everything that happened previously. It does not look at what happens further in the function. In non-trivial functions, this results in a lot of careful code analysis and means that each statement needs to be treated uniquely in space and position. Flow in-sensitive analysis the entire block of code inside a method before calculating operations on these values. Instead of having to analyse the code sequentially and take into account changes, flow in-sensitivee analysis can do it quickly and efficiently.

One of the big differences between flow sensitive and flow in-sensitivee analysis it is possible to update the points to set (the value currently stored by a variable) when assignments occur in code. In theory this means that at any one point of the code the set of values pointed to by a pointer is a single value. When an assignment happens this value is updated. Unfortunately, this does not always happen. Take for instance a piece of code where an IF statement assigns one value to X in the true case and another value to X in the false case. As it is un-decidable the result of the IF both values may be pointed to after the statement. In the case of flow in-sensitivee analysis the entire set of possible values X could store are in the set throughout the function analysis.

Line	$flowsensitive$	$flowin-sensitivee$
3	a:= {0} b:= { }	a:= {0,1} b:= {0,1}
4	a:= {0} b:= {1}	a:= {0,1} b:= {0,1}
6	a:= {1} b:= {1}	a:= {0,1} b:= {0,1}

Table 2.1: Explanation of point's to value of figure 2.5 (page 22)

```
1  int funcA( void )
2  {
3       int a = 0;
4       int b = 1;
5
6       a = b;
7
8       return 0;
9  }
```

Figure 2.5: Values known at each point in code

2.4.2 Context sensitive vs. Context in-sensitive

A very complex issue, especially with regards to function pointers is that of context sensitivity. Context sensitivity deals with the current environment under which a function is called. In the case of a context sensitive analysis the parameters with which a function is called are treated independently. In the case of context in-sensitive analysis all calls to the same function are exactly the same irrespective of the parameters.

```
1  int funcA( void )
2  {
3      int a = 0;
4      int b = 1;
5
6      funcB(a);
7      a = b;
8      funcB(a);
9
10     return 0;
11 }
```

Figure 2.6: Values known at each point in code

Line	*Context − sensitive*	*Context − in − sensitivee*
6	Parameter := {0}	Parameter := {0,1}
8	Parameter := {1}	Parameter := {0,1}

Table 2.2: Explanation of function calls values of figure 2.6 (page 22)

2.4.3 Field sensitive vs. Field in-sensitive

Field sensitivity deals with structures and arrays in a C program and the associated fields. If a structure contains pointers a, b if the analysis is field in-sensitive a call to either is

treated as a possible call to both. In the case of field sensitive analysis calls are handled individually. This particular analysis choice can make a huge difference with respect to the analysis time and accuracy.

2.4.4 Directional vs. Symmetric

The final consideration that needs to be taken into account when analysing code is only necessary for flow in-sensitivee analysis. This situation occurs when one variable is assigned the contents of another. If variable A can point to x and variable B can point to y then the statement A = B appears. In symmetric analysis it is thought that A may now point to y and B may now point to x. In directional analysis it is only thought that A may now point to y.

```
1  int funcA( void )
2  {
3      int a = 0;
4      int b = 1;
5
6      a = b;
7
8      return 0;
9  }
```

Figure 2.7: Values known at each point in code

Line	*Directional*	*Symmetric*
6	a:= {0,1} b:= {1}	a:= {0,1} b:= {0,1}

Table 2.3: Explanation of point's to value of figure 2.7 (page 23)

2.5 Methods for Extraction

Though the above factors need to be taken into consideration when constructing a call graph, the choice of factors and means of implementation can play a significant factor in the final result. For the remainder of this section a discussion on a number of pointer analysis algorithms will be discussed and their benefits and short comings. Extraction of the function calls which are made via a direct call will not be discussed, as these can be trivially read off from the source code.

2.5.1 Context-sensitive Analysis

Emami's Algorithm

Emami's solution for the pointer aliasing problem is a flow sensitive solution [EGH94b]. It is also context and field sensitive, due to the solution being flow sensitive: the directional and symmetric attributes do not apply. The main concept behind this algorithm is in the use of points-to pairs. These are explained as; If a variable x points to a variable y at some point p_1 then we know that x must contain the address of y. If later at some point p_2, x is assigned to v it is known that x no longer contains the address of y, but now contains the address of v. A benefit of this method is that it is possible to cross off earlier relationships and thus maintain a good idea of what a variable contains at some execution point of the program. Being able to cross off previous 'points-to' relationships is known as a strong update and is a major difference between flow-sensitive and flow in-sensitive analysis which uses weak updates [HP97].

The process of extracting the points to set using Emami's algorithm is as follows: Begin by perform a depth first search of the application starting from the main function. For each method compute the invocation graph, the invocation graph is the graph of what occurred, and which functions were called in a given function. This can then be used to form the points to set for that particular function. There are two special cases when dealing with this procedure. Firstly, when a function call is made to a function that has been processed already or currently being processed the search is stopped. This situation occurs when recursion is present. The end function node is marked as an approximate node, and the previous node is marked as the recursive node. Secondly, when the search encounters a function pointer call, it calculates the *points-to* set up to that point. Once this set is created, the search iterates through all possible function calls and creates invocation graphs of these. Once this has been completed the search continues.

Empirical analysis shows that it does give a good approximation of the call graph [EGH94a, HP97]. It is interestingly noted that flow sensitive algorithms tend to achieve similar results and that the graphs are roughly equivalent. Another interesting point is that applications of over 30,000 lines of code were tested. With the above overheard of context-sensitive analyses one would think that the search would struggle. From the testing carried out this does not appear to be the case, context-sensitive analysis handled the code efficiently.

2.5.2 Context-insensitive Analysis

In the world of flow in-sensitive pointer analysis there are two major methods, Andersen and Steensgard [And94, Ste96]. All other algorithms of this type can be thought of as a variation of these two main ideas.

Variation	*Andersen*	*Steensgard*
Flow-sensitive	False	False
Context-sensitive	False	False
Field-sensitive	False	False
Direction	Directional	Symmetric

Table 2.4: Variations in Context-insensitive Analysis

Andersen's Algorithm

Andersen's solution for the pointer aliasing problem is a flow in-sensitive solution [And94]. It is context insensitive and field sensitive. An interesting property of Andersen's Algorithm is that it is directional. The main concept behind this algorithm is the use of pointer assignments as constraints on the analysis. This can be explained as follows;

Andersen begins by defining a pointer abstraction. In this context, a pointer abstraction is a map from some abstract location (variable) to a set of abstract locations (Memory). The algorithm works by analysing the entire function and coming out with a set of abstract locations such that: Each variable in the function is inside the set of computed abstract locations. From this value it is possible to lookup the set of all possible values that the variable may, or will point to. How are these pointer abstractions generated? The view that is taken by Andersen is that there are constraints on the system such that it can be inferred if a value may point to or be assigned to a variable. As the initial relationships are formed, more relationships can be seen to exist between variables. Such that if a value in the abstraction locations of a pointer abstraction, in turn points to its own set of abstract locations a relationships exists between the original pointer abstraction and the final abstract locations.

In a more concrete example [Str06]. Take the initial pointer assignment as the base constraint such that we now have a pointing to the value b. In the second line of code there is an assignment such that b now points to c. Thus a now points to b and in turn to c. In the fourth line d is pointed to e. Thus a new points to set is formed. In the final line, a is assigned to d. Thus the points to set of a now contains: b pointing to c and d pointing to

```
1  **a, *b, c, *d, e;
2  a = &b;
3  b = &c;
4  d = &e;
5   a = &d;
```

Figure 2.8: Code to generate Andersen's Points To Graph

e. See figure 2.9 (page 26) for the point's to graph generated.

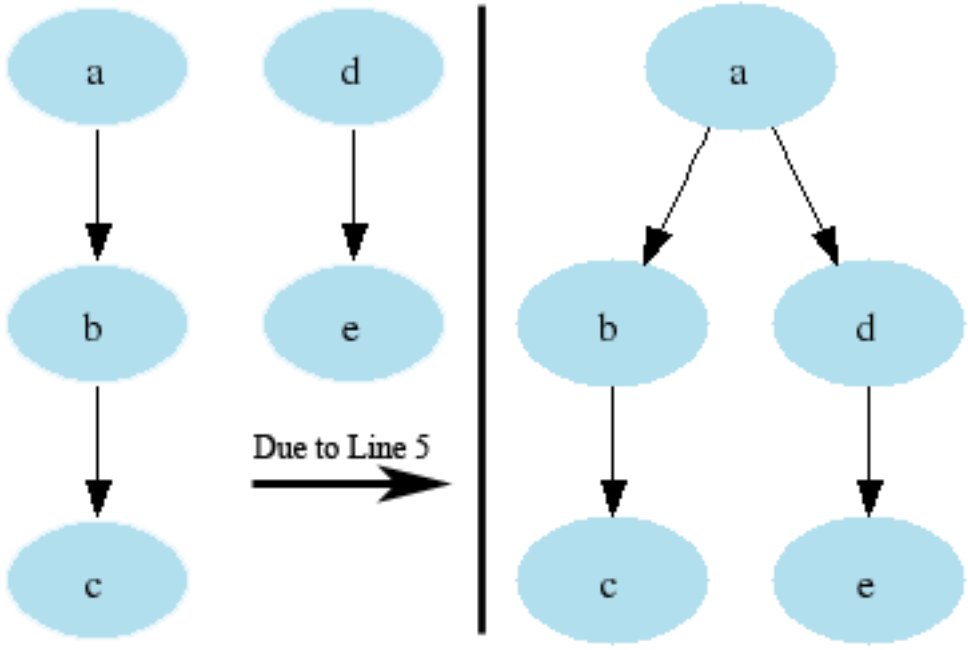


Figure 2.9: Points to graph generated from Figure 2.8 (page 26)

As can be seen by the above diagram, the result of Andersen's analysis is a number of interconnecting nodes. Each variable is located in a single node and has edges to other nodes. Each edge is a constraint which is formed when indirections are made. Thus the pointer abstraction can be understood from the graph, Figure 2.9 (page 26).

After the analysis and generation of the set of abstraction locations further analysis can be carried out on the function. When encountering a function call the constraints which were previously employed to get the abstraction locations are now applied to the parameters passed. In the situation when a functions return value is assigned to a value the constraints are also applied to the return values.

Andersen's algorithm is the slowest of the context sensitive analysis methods taking $O(n^3)$ to compute the pointer abstractions for some program. There exists |n| nodes in the system, where n is the number of variables, and as such at worst case there are $O(n^2)$ edges. Thus when we compute the fixed point computation to calculate the *points-to* set it is computing the transitive closure of a dynamic graph and as such takes $O(n^3)$ [Str06, Ryd79]. Though it can take a long time to compute the result, Andersen's algorithm is the most precise of the flow in-sensitive algorithms.

Steensgard's Algorithm

Steensgard's solution for the pointer aliasing problem is a flow in-sensitive solution [Ste96]. It is context insensitive and field insensitive. An interesting property of Steensgard's Algorithm is that it is symmetric. The main difference behind this algorithm and Andersen's is that Steensgard uses type constraints rather than term constraints. This can be explained as follows;

When a variable is initialised it is associated with a second type, where the type is the set of locations that the variable can point to. This type can then be used in equalities to monitor which variables point to what. As such when an assignment occurs we know that the types (Points-to set) must be the same. Thus the point's to set of both sides of the assignment are joined together in a new type. Thus the program will remain well typed by joining each variable together.

In a more concrete example [Str06], take the initial pointer assignment as the base constraint such that we now have a pointing to the value b. In the second line of code there is an assignment such that b now points-to c. Thus a now points-to b and in turn to c. In the fourth line d is pointed to e. Thus a new points-to set is formed. In the final line, a is assigned to d. The result is quite different from Andersons. In Steensgard's algorithm, the types of the values that a now point's to are the same. Thus b union d. Now that b and d are the same type and both point c and e are it implies that c union e; hence a single set. See Figure 2.11 (page 28)for the points-to graph generated.

As can be seen from the above code Steensgard's technique does not give a very precise answer. Every time an assignment happens information as lost with the union. The benefit of this technique is that the points-to set can be resolved in linear time O(N)

An implementation of this algorithm is based around the Union Find data structure. When a new pointer is created a type set is created. Each time an assignment happens, a

```
1  **a, *b, c, *d, e;
2  a = &b;
3  b = &c;
4  d = &e;
5  a = &d;
```

Figure 2.10: Code to generate Steensgard's Points-To Graph

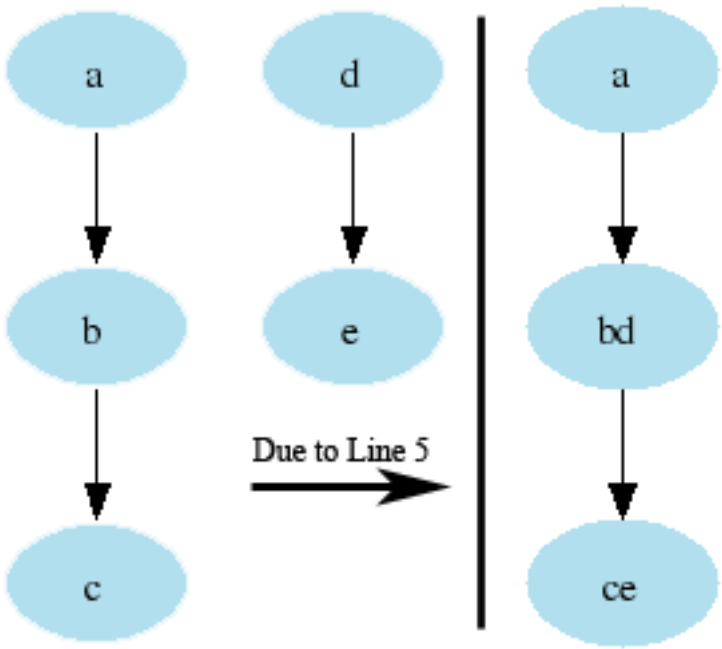


Figure 2.11: Points-to graph generated Figure 2.10 (page 28)

union of sets is performed. There are two main methods, FIND which locates the current points-to set of a type and the UNION method which joins a type together. Sets can be represented by inverted trees, node points-to parents and root node is the set. As noted above this method is much quicker, but results in reduced accuracy.

2.5.3 Comparison

A lot of empirical testing has been performed between these two algorithms [Str06, Das02, And94, Das00]. The conclusion of most tests is that in small programs, time isn't really a factor at both algorithms finish in similar time. The graph generated by Andersen's is invariably more precise than Steensgard's due to the extra analysis techniques. As programs get bigger the complexity starts to become a problem. In a number of tests carried out on GCC which contains 148,000 lines of code, the average size of the 'points-to' set extracted by Steensgard was 245.8 points being dereferenced. In the same situation the average number of points-to sets in Andersen's analysis was 7.71. As the above results indicated, the Steensgard analysis technique returned a much less accurate overview of the application.

As a contrast the Perl compiler was analysed. This program contained 23,700 lines of code, much less complex than GCC. The results show that the average points-to set returned by Steensgard had 36.1 dereferences and Andersen's had 22.22. Thus the difference was not nearly as pronounced.

There are quite a number of algorithms that try improve this data by various methods. Though none make a huge difference. Attempts have been made to combine these two analysis methods with some success [Atk04, Bur01, Das00, God02, Hin01, Das02, Str06, Ryd79, Sun03, ZRL96, GDDC97, MNGL98, MRR04].

2.6 Other Techniques

2.6.1 Real time analysis

The above algorithms deal with the extraction of a call graph directly from the source code of an application. A second method of extracting the function calls exists and does not rely on source code analysis. The other technique is the extraction of the call graph while the application is running. Thus at the run time of an application, the function calls and their calling context are noted. When the analysis is complete, a complete call graph will

be extracted of the program run.

How is this useful? As described earlier, Steensgard's and Andersen's algorithms were used to analyse GCC, a large application. The reason for GCC being so large is that it has many features, some which are not used in everyday use. If someone wanted to understand a specific path that the program executed to achieve a task, this static analysis would result in extra data not relating to the problem. A better solution would be to only extract the function calls from a specific application run. Analysis of the function calls during the run results in a sub-set of the applications calls. These calls directly relate to the goal being checked. As this technique relates more to being able to capture state at discrete intervals (function calls) of an applications run there is not as much academic literature on the topic. Fortunately, this idea of following the flow of data is built into the aforementioned tool, GCC [Fou07, Hub04].

It is possible to compile a C application using GCC with a specific flag such that every time a function is called a *profiling* function is called [Jon05]. This parses the address of the function being called. When a function exits another *profiling* function is called. Thus it is possible to store to file the memory addresses of entry and exits of functions calls. Once the application has completed execution the call graph of the applications run has been extracted. Using another utility Addr2line can now translate the memory address calls of functions to function names.

One problem still exists, if the application performs the same task multiple times, the extracted call graph will contain multiple call chains which are exactly the same. Before having a complete call graph this duplicate information should be reduced and turned into a more compact version for graph generation. Thus a call graph has been extracted from an application during the applications runtime. [Fou07, Jon05]

2.7 Graph Creation

The previous section discussed the techniques for extracting the flow of an application from the source code; the second stage in generating call graphs is in the creation of acyclic diagrams. The following is an overview of some techniques and existing systems to generate these graphs.

2.7.1 Introduction

A graph can be used as a way of showing relationships between data. A graph, G, consists of elements known as vertices and edges. (Note: vertices are often referred to as nodes). Edges contain two endpoints which connect vertices together. When a graph is given some direction it is known as a directed graph (digraph). The size of a graph is defined as the number of edges in the graph and the edge which connects to vertices x and y is denoted xy. Loops are defined in graphs by edges where both endpoints lie on the same vertices. As a call graph is a sequential analysis of an applications flow, the graphs that will reflect the application will be directed; where the direction is the calculated application flow.

To formally define a directed graph G, we say that G is an ordered pair G = (V, A) where V is the set of vertices and A is the set of ordered pairs of vertices, these are also known as directed edges. An edge e = (x, y) is directed from x to y if the path leads from x to y, figure 2.12 (page 31). In this case x is known as the predecessor and y is known as the successor.

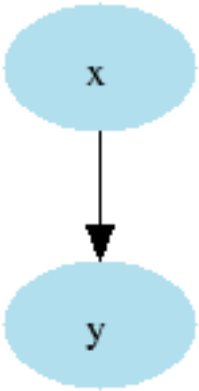


Figure 2.12: Simple directed graph

2.7.2 GraphViz and DOT

Graph Visualization or GraphViz [GN00, ATT07] for short is a collection of applications for creating graphs. GraphViz is distributed under the Common Public license V1 and was originally created by AT&T at XEROX PARC Alto.

Designed as a multi-purpose tool, it is very flexible and has applications both in database design, software creation and networking. The application has been ported to

both Windows and Linux. Graphs are defined using the DOT language [ATT06]. This language is quite descriptive and can be used to describe directed graphs correctly. GraphViz outputs vector based images (SVG) that preserves quality

One of the most compelling features of this application is that there exist many applications that use the GraphViz output and DOT file format to display information in a very rich way. One of the most interesting of these is ZGRViewer. The ZGRViewer is a Java based application that can take GraphViz applications and display them in an interactive environment. The application is based on the Zoomable User Interface (ZUI) toolkit which is Java based. The idea behind the ZUI is to add functionality to the Java swing libraries for common problems such as:

- Allowing zooming
- Allow movement of image
- Object animations

Probably one of the biggest uses of this system is in the NASAmak Blue Marble earth simulation. This allows the view images of the Earth and zoom in on them [NAS06]. Using the GraphViz platform a number of applications for generating graphs were created. The most well known of these was Acacia .

2.7.3 Acacia

Acacia is a composition of a number of AT&T research projects [God02]. The aim was to create a tool that could extract calls graphs from C/C++ programs. Acacia was built using the following tools:

- Cia - A C extractor
- Ccia - A C++ extractor
- CQL - Query language for extracted language
- Dot - Layout engine for graphs
- Ciao - Interface for application

The process of extracting a graph from source code is as follows. Ccia/Cia would extract the information and store it in a database. CQL would then be used to query this data and extract the useful data to generate the graph. From this stage, the returned CQL data would be turned into DOT format that could be used by GraphViz to graph the application.

Cia/Ccia worked as semantic/lexical analyzer to extract information directly from the source code. Though this application does do some of leg work of extracting the call graph from the application it contains a number of problems. Querying of files is complicated and work on the project stopped sometime ago.

2.7.4 Force-Directed Graphs & JSVIZ

Most graphing algorithms are based around generating a single representation of the data. This results in a static representation of the data. One of the primary concerns with this method of graph generation is the layout of the nodes and edges. It is imperative that the data is laid out in an understandable way. Once generated it requires the creation of a new graph to lay out the nodes differently. A different solution to this is the force-directed graph [DHGN99]. Nodes and edges are assigned physical attrbitues. This includes spring in the edge and the weight of the node. The nodes and edges are then added to the graph. At this point the graph is a mess and not very understandable. What happens next is the forces repel and attract each other. This operation iterates until the nodes reach a state of equilbirum. What is left is a graph where nodes are layed out with a similar distance between them. Using this method overcomes some of the complexity in understanding graph theory [1]. The iterative nature of the algorithm also means that it can be incrementally generataed reducing some complexity.

An implementation of the force-directed algorithm is JSVIZ by Kyle Scholz [Sch06] . JSVIZ is built to run in a web-browser using Javascript. The algorithm begins by adding nodes to the plane. These nodes then repel against each other. Shortly after this edges are added between nodes. The edges constrain the nodes and they eventually reach an equilibrim. Unforuntanely JSVIZ does not currently support directed graphs. Currently every edge in is undirectional. It is conceivable that JSVIZ could be modified to support directed edges.

[1] Complex algorithms used in planar graphing can be ignored

Chapter 3

Extraction Framework

3.1 Introduction

So far different techniques to aid in the process of call graph extraction have been discussed see section 2.5 (page 23). Unfortunately these methods do not give the means to extract call chains directly from source code. Instead they are concerned with analysing specific abstractions in C, pointers. Thus the question is raised, how can one extract a call graph from an application? Before answering this question, a little knowledge of compiler theory will come in useful.

3.2 Conversion to Abstract Syntax Tree

The syntax and semantics of a program language, such as C, are non-trivial and at times very complex. This is well illustrated by the many ways of calling functions. It would be nigh on impossible to directly extract meaning from the source code and even harder to write an algorithm to do this. A popular programming construct is that of the binary tree. There are many well known techniques for extracting information off a binary tree. Various algorithms such as pre-order or post-order traversal are well known. In addition searching a binary tree is at worst O(log n) [Bla07]. A representation of the program as a binary tree would be ideal and provide a good framework to begin the extraction process. To convert source code into a binary tree requires using parts of a compiler. These are:

- Lexer

- Parser

An initial compilation process sufficient to produce the binary tree is as follows:

A lexer understands the syntax of the chosen programming language. As such it is concerned with matching syntax from raw source code. To go about matching this syntax the lexer has a set of rules, known as regular expressions. When a piece of source code is passed to the lexer it attempts to match the input against these rules. On matching input to a rule the lexer creates a new token. This token is a representation of the source code in memory. The lexers job is complete for now. The newly created token is now passed to the next compilation stage, the parser. A parser understands the semantics, or meaning, of the language. Thus the parser attempts to take tokens from the lexer and understand where they fit into the already parsed code. Like the lexer the parser has a set of rules. Unlike the lexer; rules now match patterns of tokens rather than source code. The lexer and parser work together converting source code first to tokens and then checking that the token is valid in the context, and thus semantically correct. How are nodes added to the binary tree, known as an abstract syntax tree (AST)? Nodes are added when the parser matches a token to the rule, this occurs when a valid piece of code has been found. This token is then added to a binary tree in the appropriate position. Thus when the lexer has finished what is left is the AST.

3.3 Representation

The following are a selection of the most important abstract syntax trees used in the call graph extractor:

1. Variable Declaration
2. Function Pointers
3. IF-Statement
4. Function Declaration
5. Function Call

3.3.1 Variable Declaration

Looking at figure 3.6 (page 38) the lexer would first match the *int* characters. This would then be created as a token and passed to the parser, a set of rules would be matched that corresponded to variable declaration. The next characters matched would be *value*

associated with the integer declaration. This would result in the parser matching the *int* type with the name *value*. At this stage the parser knows that a variable of type int has been declared. When the semi-colon is hit the code has all been processed leaving the AST in a complete state, figure 3.5 (page 38). Though this example is relatively simple it does give the basic knowledge one needs to understanding the rest of this dissertation (See Appendix for complete Lexer and Parser Rules).

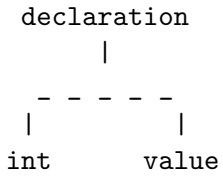


Figure 3.1: Abstract Syntax Tree for variable declaration

3.3.2 Function Pointers

Function pointers are broken into a set of two declarations. The first part of a function pointer is the return type. This is put to the right hand side of the AST. The section of the declaration deals with the variable being defined. The right hand side takes any arguments that the function takes, the left hand side defines that the declaration is a pointer and the name of the pointer. This double declaration syntax is used for all pointers in the AST.

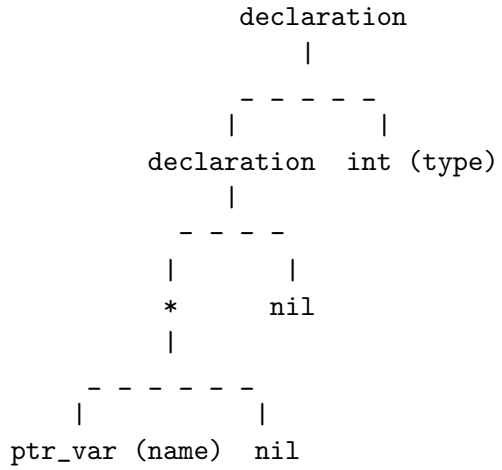


Figure 3.2: Abstract Syntax Tree for Function Pointer declaration

3.3.3 IF-Statement

The abstract syntax tree for an IF-statement starts with an *if* at the root. On the left hand side of the IF the conditional statement is placed, in this case the integer 1. On the right hand side is the body of the IF statement.

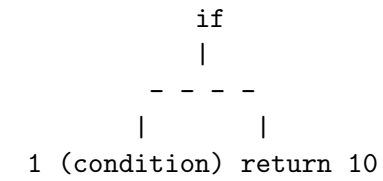


Figure 3.3: Abstract Syntax Tree for IF-Statement

3.3.4 Function Declaration

The function declaration is broken up into two sections. On the left hand side the formal parameter declarations are placed to the right. The return type of the function is placed on the left hand side. On the right hand side of the root there are two more parts to the function. On the left is the name of the function and on the right is the body of the function.

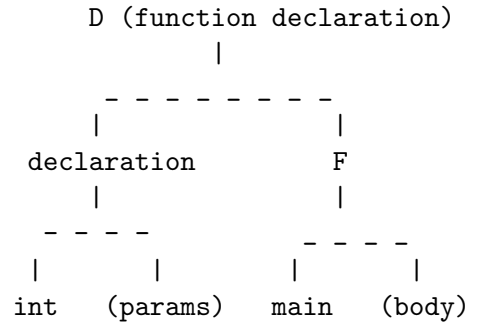


Figure 3.4: Abstract Syntax Tree for function declaration

3.3.5 Function Call

A function call is designated by the apply keyword. On the right of the apply is the name of either the function or the function pointer being called. On the right hand side is list of

any actual parameters passed to the pointer. Now that a framework exists for building our algorithms, the AST, it is possible to begin creating the various call chain extractors.

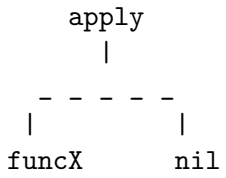


Figure 3.5: Abstract Syntax Tree for function call

```
1  /* Variable Declaration*/
2  int value;
3
4
5  /* Function Pointer Declaration*/
6  int (*ptr_var) ();
7
8
9  /* IF-Statement t*/
10 if(1)
11 {
12     return 10;
13 }
14
15
16 /* Function Declaration */
17 int main()
18 {
19     /* Body  */
20 }
21
22
23 /* Function Call */
24 funcX();
```

Figure 3.6: Code listing

3.4 Call graphing & Code Compilation

The previous section outlined a number of abstract syntax trees required for call graphing. It is interesting to note that there are many symmetries between the AST generated for code compilation and the one used for call graphing. The general structure and process for creating the AST is the same; both using a parser and a lexer. Though this similarity

exists the trees do not necessarily need to be the same. When generating machine code all details needs to be represented in the AST. For call graphing a simplified version of the AST can be be used.

To what extent could the AST be simplified? In theory the compilation stage could perform the call graph extraction! This could be achieved by only recognising function declarations and invocations. The call graph could then be read straight off the tree. As will be seen in Chapter 4 (page 40) this simple solution reduces the accuracy of results and leads to an incorrect calls graph.

Chapter 4

Extraction Algorithms

4.1 Call Graphing using Naive Analysis

To aid readability and understanding of a program the logic is often divided into smaller chunks. These small chucks are known as functions. As the call graph concerns itself with the interplay between these functions it seems prudent to start building our algorithm with these in mind.

4.1.1 Aims

The aims of the naive algorithm is to avoid any undue complexity. Thus when presented with two solutions, the simpler of the two should be chosen [1]. The second aim of this algorithm is to form a base for improvements, or extensions, for extension in latter algorithms. The final aim of this algorithm is to limit the logical complexity of call graph extraction with a goal to complete in near linear time.

4.1.2 Algorithm

Inter-Procedural Analysis

Before diving straight into analysing blocks of code it would be a good idea to get an overview of the programs structure. One way of doing this would be to extract all the

[1] Simpler with respect to the results of the algorithm

functions in the program. Thus the first part of the naive algorithm extracts the functions and their bodies from the AST. Now that all the functions contained in the program have been found the question of how to go about extracting calls from these functions bodies arises. The simplest solution is to treat every function body as an independent block of code, this is known as inter-procedural analysis. For this process to work the algorithm must presume that each function does not directly affect any other function [2]. Once every function has been analysed the results can be interpreted independently.

Call Context & Conditional Statements

Now a method for processing each function has been decided on. The next step is to analse the functions body. Firstly what should the code do when it hits a function call? As this algorithm aims to be as simple as possible it will not use any of the pointer analysis algorithms outlined in the literature review. The algorithm must treat each call as a direct call. Now that function calls have been dealt with it is possible to address the context in which a function call is made. Syntax such as conditionals are ignored as they add complexity and diverge from the aims. Thus the algorithm cannot tell under what context a call is being made. In conclusion the algorithm also does not concern itself with the possibility that a function *may* or *will* be called.

Summary

What kind of algorithm does this leave us with and how complex is it? Using the terminology outlined previously this algorithm is inter-procedural, context in-sensitive [3], flow in-sensitive, field in-sensitive. The algorithm works in two main stages and is built around a pre-order tree traversal. Visiting every node in a pre-order tree traversal is O(n) thus fulfilling the aim for the algorithm to finish in linear time [4].

Pseudo-Code

The algorithm can be outlined as follows. Look at each function in the program sequentially. For each function look at the statements located in the body. When a function call is found,

[2] For instance by passing a function pointer as a parameter to the function

[3] Context in-sensitive also applies to the methods of updating pointers in the point-to set. In this situation we are referring only to the context in which a function is invoked

[4] Extracting the individual functions from an AST is always a single tree walk at worst O(n). O(n) + O(n) ≡ O(2n) ≡ O(n)

add the name of the function being called to the list of called functions. If the statement is not a function call ignore it and look at the next statement. When every function has finished being processed what is left is the call graph for the application. A pseudo-code version of this can be seen in figure 4.1 (page 42).

```
1  startNaivelExtraction()
2  {
3       functions = get_functions( ast )
4
5       foreach(Function f in functions)
6       {
7           extractCalls_Naive( f )
8       }
9  }
10
11 extractCalls_Naive(AbstractSyntaxTree ast)
12 {
13      if(ast->type == FUNCTION_CALL)
14      {
15           AddFunction( ast->name )
16      }
17      else
18      {
19           extractCalls_Naive( ast->left )
20           extractCalls_Naive( ast->right )
21      }
22 }
```

Figure 4.1: Algorithm for Naive Call Extraction

4.2 Call Graphing using Steensgard's Analysis

Though the naive algorithm works well with respects to its aims, it has a number of shortcomings. Probably the most significant of these is that it does not recognise function pointers. Function pointer calls are not checked for and as such are treated as direct function calls. This becomes a problem when a function variable is called; the variable name is treated as the name of the function. As one may guess this leads to a non-existent function being added to the callee list. As a corollary, functions that may be called are not represented in the call chain; instead the function pointers name would be listed. A solution to this problem is required. Outlined in the literature review were two code analysis techniques for extracting the points-to set from a block of code. These algorithms are used to track the use of pointers. The first algorithm, devised by Steensgard, can be added to the naive algorithm without major changes. This is due to the algorithm being inter-procedural.

4.2.1 Aims

The aims of this algorithm are to expand on the base provided by the naive algorithm. As noted above one of the major flaws of the naive algorithm was its simplistic treatment of pointers. The second aim of this algorithm should be to add the facility to recognise and monitor the use of function pointers.

4.2.2 Algorithm

Abstract Types

As this algorithm is built upon the naive algorithm it begins in a similar fashion. The first stage is to extract each function and then sequentially process them. Once again this will allow the results to be collected at the end, independent of other functions (inter-procedural analysis). Things start to get a little bit more interesting when interpreting each function body. Steensgard's algorithm begins by giving every variable present in a function a unique abstract type; independent of programming language, see section 2.5.2 (page 27). This adds an extra step the the analysis stage. Before extracting function calls the function body is searched for variables. Each variable declaration results in a new associated abstract type. These types are added to a data structure known as the Union-Find data structure, see section 7.3.1 (page 65). Function extraction can now begin.

Function Calls

The body of the function is now searched for calls. In contrast to the naive algorithm when a function call is found, it is not automatically added as a direct call; it is instead processed further. This extra processing tries to match an abstract type with the value being called. If no match is found then this call is to an inexistant type; instead it is a direct call. If a match is found then the call is via a function pointer. The abstract type is now marked as having been invoked. Intuitively one would think to extract the functions that are pointed to now, instead the processing continues with the next AST statement. The reasons behind this will become apparent shortly.

Pointer Assignment

Though it is possible to mark types as having been called there is one important piece of the puzzle missing. The question of how assignments to our abstract types are handled. Before proceeding with this the reader must realise that there are two types of assignments that may happen. Firstly the assignment of a function to a pointer. This now implies a call to the pointer is the same as a direct call to the function. The other case is when a pointer is assigned to another pointer.

Steensgard's algorithm does not provide a solution to the first case explicitly and as such it is left to the author to think of a suitable solution. To aid in finding a solution to this problem it is worth forgetting that the value being assigned to the pointer is a function . As it is known that the value being assigned to is directly to a function [5] it is sufficient to add the function to the set of functions pointed to.

In the second case Steensgard provides an exact process for assignment. The assignment of two variables can be thought of as moving from two types into a new unused third type. This ad-hoc type creation is achieved by a union of the values stored in the types. One might be inclined to wonder what happens to the functions that were assigned to the original types. As the transformation from two types to one is based of unification all the meta data associated with each abstract type is retained. More interestingly the information that was pertinent to either type is now applicable to both types; or in actual fact the new single type that represents them. See Union-Find data structure section 7.3.1 (page 65) for further implementation details.

[5] This can be trivially noted by the layout of the AST

This process of type redefinition and assignment is continued until the entire function body has been processed. What is left is a call graph that describes the code as a whole.

Summary

What kind of algorithm does this leave us with and how complex is it? Using the terminology outlined previously this algorithm is inter-procedural, context in-sensitive, flow in-sensitive, field in-sensitive. The algorithm works in three main stages and is built around a pre-order tree traversal. Visiting every node in a pre-order tree traversal is O(n) [6]

Pseudo-Code & Algorithm Logic

To further illustrate the steps using Steensgard's algorithm the algorithm logic can be seen in figure 4.2 (page 46). A pseudo-code version of this can also be seen be seen in figure 4.3 (page 47).

[6] Extracting the individual functions & types from an AST is a single tree walk each at worst O(n). O(n) + O(n) + O(n) ≡ O(3n) ≡ O(n)

1. Defining Types: Iterate through the statements of the function being analysed
   (a) When a pointer variable is found, create a new distinct type { T_1, ... , T_n }
   (b) Add type to list of types present in function
2. Process Body: Iterate through the statements of the function being analysed again
   (a) Check if the current token refers to an assignment. If so enter one of two stages depending on the right hand side value:
      i. Function: In the case that the right hand side is a function name, add the function to the list of functions the pointer points-to
      ii. Pointer: In the case that the right hand side is a pointer, find the type associated with each pointer. Proceed to union the two types together to make a third new type $T_1 \bigcup T_2 = T_k$
   (b) Otherwise check if the current token refers to a function call:
      i. Loop through the currently allocated types looking to see if the name of the call refers to a type. Mark the type as invoked
      ii. If the previous condition was not matched then this is a direct function call. Add it to the list of direct calls
3. Extracting Graph: Once the process has completed there are two main sets of data created
   (a) The set of direct function calls: These can be read straight off
   (b) The set of remaining types. To extract the calls use the following process:
      i. Check to see if the type, T_y, was marked as invoked. If invoked add all functions referenced in T_y the list of direct function calls
      ii. Check the next type, T_{y+1}

Figure 4.2: Algorithm for Steensgard's Call Extraction

```
1  startSteensExtraction()
2  {
3        foreach(Function f in functions)
4        {
5             createVariables_Steens( f )
6             extractCalls_Steens( f )
7        }
8  }
9
10 extractCalls_Steens(AbstractSyntaxTree ast)
11 {
12       if(ast->type == FUNCTION_CALL)
13       {
14            calculateCall( ast )
15       }
16       else if(ast->type == ASSIGNMENT )
17       {
18            assignValue( ast->ASSIGN_TO, ast->ASSIGN_FROM )
19       }
20
21       extractCalls_Steens( ast->left )
22       extractCalls_Steens( ast->right )
23
24 }
25
26 calculateCall(AbstractSyntaxTree ast)
27 {
28       foreach(variable in FUNCTION)
29       {
30            if(ast->type.name == variable.name)
31            {
32                add_variable(ast->name)
33            }
34            else
35            {
36                add_function(ast->name)
37            }
38       }
39 }
```

Figure 4.3: Pseudo-Code for Steensgard's Call Extraction

4.3 Call Graphing using Emami's Analysis

The algorithms discussed so far do a good job of analysing calls between functions; each having its own benefits. The first one presents a linear time algorithm and the second adds the capability to analyse function pointers. Unsurprisingly there have been a number of necessary simplifications. The ordering of statements inside the function body has no effect in either algorithm. Thus the context of a function call is ignored. This leads to the conclusion that currently it is impossible to know if a function *may* or *will* be called. Secondly, the flow of assignments is ignored; leading to only weak updates, see section 2.4.1 (page 21). Being able to ascertain the context of a function call and the flow of pointer assignments would lead to a call graph that presents a truer reflection of the code body. Finally, the effects of function calls and their associated call chains should be analysed. The second algorithm for points-to set extraction devised by Emami presents a solution to these problems, see section 2.5.1 (page 24). Major changes to the naive algorithm are required; unlike previous algorithms Emamis is intra-procedural.

4.3.1 Aims

The aims of this algorithm are to create an algorithm that performs intra-procedural analysis. The second aim of this algorithm should be to recognise if a function call *may* or *will* be called. Finally the algorithm should provide a more precise method of pointer analysis; allowing for strong and weak updates.

4.3.2 Algorithm

Intra-Procedural Analysis

The naive algorithm presented an inter-procedural solution; processing each body of a function independently. One of the aims of this algorithm, inherent in the Emami algorithm, are the flows between functions being analysed, known as intra-procedural analysis. Another method for processing functions is needed. Before explaining the method it is worth thinking about the steps used to create an interpreter [RLV+96]. This method begins by selecting some arbitrary function and processing it. To proceed to a new function, a call needs to be made to it from the current body. Parameters are mapped to the new function (see section 7.3.2 (page 67)) and the processing begins again. This continues until the full body has been processed. When every statement in the callee functions body has com-

pleted; control is returned to the caller. Unlike an interpreter the code is not evaluated, rather each statement in the function body is analysed. The question of which function to begin processing on can be dealt with relatively easily. All programs in C contain a main function and as such it is the logical choice.

The more observant may have noticed a problem with this solution, recursion. Generally recursive functions have an exit condition that is eventually met. As the algorithm does not concern itself with evaluating conditions an infinite loop will occur (via direct or indirect recursion). The solution to this problem is as follows. When a function call is made, the algorithm checks through the previous functions in the call chain. If the function being called is not present continue as normal. If the function is found mark the current function as the end node and the pivot function as recursive.

Conditional Statements

Now that a method has been outlined for proceeding to analyse each function it is time to look into the changes that occur at the analysis stage. Firstly this algorithm is flow and context sensitive. Thus it is important to know whether or not a statement is inside of a conditional. It is worth having a think about what constituents a conditional statement. The most widely known and used conditional is the IF-statement. Limiting the conditional statements inside the IF to a single statement means that it will always get executed [7]. Depending on the outcome of the condition the body *may* get executed. In all situations the statements inside the IF body always fall into the *may* be executed bucket. As a corollary to this the question of what happens when there exists an ELSE statement in the code? The block of code inside the ELSE can be treated as though it were too inside a conditional, as for all purposes it is [8]. Following on from this FOR, WHILE, SWITCH and all other flow control statements can be treated in the same way. This realisation leads us to the conclusion that the AST can be simplified to show either the condition or the conditional body and not worry about specific language based constructs. All of these statements are just variations on the same theme. Finally nesting of conditional statements has one effect on the set these inner statements falls into, all will fall into *may*, even those in the condition. To designate if the processing is about to proceed a *may be executed* flag is set, this is reset at the end of the conditionals body.

[7] The reasons for this simplification are to avoid to issues associated with lazy order evaluation
[8] The condition is just the inverse of the IF

Assignments

Now there exists a means for recognising if a given statement *may* or *will* get executed. From here it is possible to add context and flow sensitivity. To quickly re-cap, strong updates are those where a value will be replaced by a new one. Inversely weak updates are those when it is not know if the assignment overwrites a previously held value. To understand the relevance of this it is worth looking through the cases when this occurs. Firstly if a direct function call is made inside a conditional it *may* be called. In the cases when a direct call is not inside a conditional it *will* be called. Now that the base cases have been covered it is time to dip into function pointers. If an assignment occurs and is made outside of any conditional this results in a strong update, all the previous values being overwritten. If the assignment occurs inside of a conditional the update is weak, the new assignment is appended to the list of already assigned functions or pointers.

```
1  int main()
2  {
3        int (*ptr_fX)();
4
5        ptr_fX = &funcX;          /* Point P1 */
6
7        if(VAR == TRUE)
8        {
9              ptr_fX = &funcY;    /* Point P2 */
10       }
11
12       ptr_fX();                 /* Point P3 */
13
14       ptr_fX = &funcZ;          /* Point P4 */
15
16       ptr_fX();                 /* Point P5 */
17 }
```

Figure 4.4: Example using context sensitivity

To illustrate the above concept let's work through an example. Figure 4.4 (page 50) shows both weak and strong updates. When the program hits P_1 it is not currently in a conditional state, and no value has been assigned to ptr_fX yet. Thus ptr_fX only contains the value funcX. As the algorithm moves onto point P_2 it enters a conditional section. When the value funcY is assigned to ptr_fX it will not definitely replace the value held in ptr_fX, thus a weak update is performed. The value funcX and funcY are now stored in ptr_fX, both in the may call set.

The final assignment happens at point P_4 the algorithm is no longer in a conditional

statement. Thus the update is strong and replaces the values held in ptr_fX. ptr_fX now only stores the value funcZ. What effect does this have on the call graph? There are two interesting cases in this example. At point P_3 the function call could be either to funcX or funcY but at point P_5 it is known that the call refers to funcY due to the strong update at P_4.

Field Insensitivity & Array Assignments

One of the caveats of Emamis algorithm is that it is field in-sensitive. An easy way of understanding the ramifications of this is to think about arrays. When a value is assigned to an array it is very hard to correctly track the index which it was assigned to. Think of the situation when a value is assigned to a random index. Thus to simplify this problem when a value is assigned the array is always treated as a normal variable but unlike other variables; updates are always treated as weak. This allows us to track assignments to arrays and other similar types, for instance structs.

Final Call Extraction

This process of monitoring whether code *will* or *may* be executed and performing weak or strong updates occurs till the final statement for the initial function has been completed. What is left is a call graph that describes each call chain in the program. These call chains are then extracted. An interesting part of this problem is a certain amount of redundancies in the call chains. As one can well imagine, a call chain could be repeated by any number of functions. Thus when extracting the final call graph from these applications this redundant information is ignored, leaving a complete call graph.

Summary

What kind of algorithm does this leave us with? Using the terminology outlined previously this algorithm is intra-procedural, context sensitive, flow sensitive, field in-sensitive. The algorithm works in two main stages and is built. Unlike previous algorithms this is not built around just a pre-order tree traversal. Instead the algorithm is built around following each call chain in the program. As each call chain could lead to all other functions in the code (One tree could lead to N trees) the code is $O(n^2)$ [9].

[9] Each tree could lead to a separate tree thus: O(n) x O(n) $\equiv$ $O(n^2)$

Pseudo-Code

To further illustrate the steps in of call graphing using Emamis algorithm the pseudo-code version of the algorithm can be seen in figure 4.5 (page 52).

```
1  startEmamiExtraction()
2  {
3       foreach(Function f in functions)
4       {
5          calculateVariables_Emami( f )
6       }
7
8       Function main = findFunction( "main", allFunctions )
9       extractCalls_Emami( main , false )
10 }
11
12 extractCalls_Emami( AbstractSyntaxTree ast,
13                      boolean conditional)
14 {
15      if(ast->type == FUNCTION_CALL)
16      {
17          calculateCall( ast )
18      }
19      else if(ast->type == ASSIGNMENT )
20      {
21          calculateAssignment(
22                        ast->ASSIGN_TO,
23                        ast->ASSIGN_FROM,
24                        CONDITIONAL)
25      }
26      else if(ast->type == CONDITIONAL)
27      {
28          conditional = TRUE
29      }
30      extractCalls_Emami( ast->left  , conditional )
31      extractCalls_Emami( ast->right , conditional )
32 }
33
34 calculateAssignment(Value TO, Value FROM,
35                     boolean conditional)
36 {
37      if(conditional == TRUE && TO->type != ARRAY)
38      {
39          assignWeak( TO , FROM )
40      } else   {
41          assignStrong( TO, FROM )
42      }
43 }
```

Figure 4.5: Algorithm for Emami Call Extraction

Chapter 5

Call Graph Visualization

Up until now the main driving force behind this dissertation has been that of extracting function calls from source code. This leaves us without a useful visualisation of the output. This section deals with various existing and novel techniques for visualising the results.

5.1 Directed Graphs

As discussed in the literature review a popular method of visualising call graphs is that of directed graphs. To quickly re-cap, directed graphs show relations among nodes via directed edges. How can directed graphs be used to describe the interplay between functions? The directed graph needs to be able to convey two bits of information. The first is the various functions that exist in the code. The second is calls between functions. Luckily there exists a trivial mapping of this information to the elements inside of a graph. Namely, nodes correspond to functions and edges correspond to calls from one function to another. As this is a directed graph each edge has a direction, denoted by an arrow. In this mapping the predecessor corresponds to the caller function and the successor corresponds to the callee. Now that the initial schema of the graph has been decided how can extra meta data be displayed?

5.1.1 Visual representation of nodes and edges

Though at first it may appear to be a trivial concept, the question of how to visualise the nodes and edges is a complex one. It stands to reason that the edge should be a line

with a arrow indicating direction on it. How should the nodes be visualised? A number of considerations must be made when designing a method for visualising the call graph. These include the visual asthetics and usual graphing standards. In this solution, nodes should be visualised by an ellipse with the name of the node, the function name, inside the ellipse's body. Something that may not be apparent when first creating these graphs is knowing where the root node is. With the addition of cycles to the graph the situation can arise where it is hard to see where the program begins. To make it clearer the initial function, generally *main*, shall be a rectangle. This allows the user to trivially understand the orientation of the graph and where the program begins.

Figure 5.1: Main calling another arbitrary function

5.1.2 Will or May Call

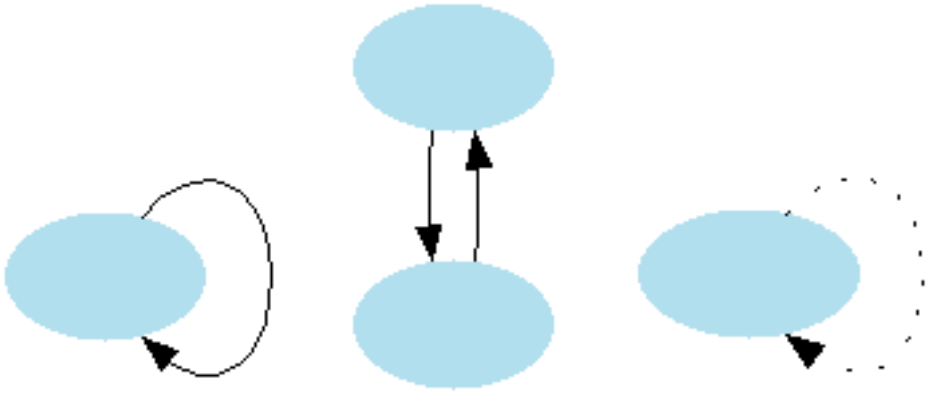
The first problem addressed is that of how to show the relationship between function calls that *may* or *will* occur. The relationship between caller and callee is already visualised using the edge of the graph. At this moment a directed edge, no matter its visual aspects, designates a call. The simplest way to show this extra relationship data would be to modify the edge. The two choices that automatically spring to mind are a solid edge and a dashed edge. From a visual perspective a dashed or broken edge would appear to the user as a possible route, whereas a complete edge would appear to be a definite route. It follows that the dashed edge should be mapped to the *may* call set and the solid line should map to the *will* call set. visualisation of this is shown in figure 5.2 (page 55).

(a) Will Call (b) May Call

Figure 5.2: Directed edges showing will or may call

5.1.3 Recursion

Following from the above; what happens when recursion exists in our graph? There are two types of recursion that occur in an application. Firstly, that of direct recursion. Direct recursion is when a function makes a call to itself inside of its own body. The second type of recursion is indirect recursion, this occurs when a call is made from some point in a call chain to a function in a previous part of the chain. Thus our graph is not strictly directed acyclic, but may contain cycles. As the cycles may either flow back and forth this cannot be thought of as a directed cyclic graph. Now that a better understanding of how recursion occurs inside our graph it is interesting to see how this manifests itself inside of a call graph. Figure 5.3 (page 55) shows a graph with a recursive node in it. This recursion falls under the first scenario as it is referencing itself. In the first graph a solid line indicates the recursion. The solid lines infer that the recursion *will* always occur, an infinite loop. The second graph shows that the call *may* occur; recursion may hopefully exit under some condition. visualisation of this is shown in figure 5.3 (page 55).

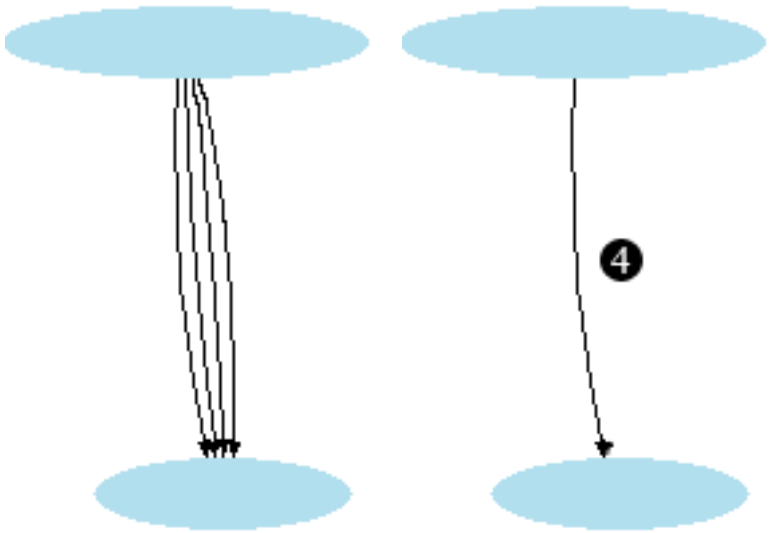


(a) Direct/Infinite Recursion (b) Indirect Recursion (c) Indefinite Recursion

Figure 5.3: Recursion shown as infinite or indefinite

5.1.4 Multiple Calls

Quite often a caller may invoke a callee multiple times inside of its body. Two ways of visualising this seem like logical solutions. The first involves adding an annotation to edges. This annotation contains a numeric value which is the number of times that the function is invoked from the caller. A small problem arises when the relation between *will* or *may* call sets needs to be shown. This can be solved by having two edges from the caller to callee, one for each call type. From a visual aspect this could be quite confusing. Initially glancing at functions could convey that just two calls occur or similar problems. A second solution involves drawing a line for each call. Every line retains it's only calling nature. Using this method it is simple to see how often a function is called and just glancing at the graph gives a good approximation. Though one may not consider it, a third solution does exist. The initial premise of the algorithm was to show an overview of the code. Possibly a better to solution would be to only show a single edge between nodes, no matter the number of calls. If *may* and *will* both exist between two nodes the stronger call, *will*, takes precedence. visualisation of this is shown in figure 5.4 (page 56).



(a) Multiple Edges (b) Annotated

Figure 5.4: Caller invokes callee multiple times

Chapter 6

Visualization of Algorithms

As can be seen from call chain extraction the various algorithms produce different results. As such it is not sufficient to choose a visualisation method that is perfect for all three. Instead the different algorithms each use variations on the schema. This section sets out to describe these. A second problem exists when visualising applications in that too much data can be displayed at one time. A method of displaying this information more succinctly is described, force-directed graphs.

6.1 Static Graphs

6.1.1 Visualisation of Naive & Steensgard's Based Algorithms

The naive algorithm is very different from Steensgard's algorithm with respects to the correctness of the results (see section 9 (page 83)). Surprisingly the variations in output metadata is the same. The reasons for this surprising result are that both algorithms are inter-procedural, context in-sensitive and flow in-sensitive. As the output of both these algorithms is the same it makes sense to use the same visualisation standard. The visual representation of these algorithms is as outlined in the previous section with one caveat. One of the major pitfalls is that neither algorithm differentiates *will* or *may* call sets. In actual fact, the algorithms see all functions as *will* calls. This is a very useful piece of information. As the *may* case can be ignored there will only exist solid edges between nodes.

The second question is how to display multiple calls? As the algorithm only has one type

of edge, definite calls, the meaning of the edges is easily clear. Thus for these algorithms there are multiple edges between the successor and predecessor to denote multiple calls, see figure 5.4 (page 56).

6.1.2 Visualisation of Emami's Based Algorithm

The metadata extracted from the final algorithm is quite different from the previous two. The reasons for this distinct difference are that this algorithm is intra-procedural, context sensitive and flow sensitive. As before, the same visualisation for edges and nodes is used. Unlike the above algorithms it is now possible to see if a function *will* or *may* be called. The presents us with a problem. In the previous algorithm it was possible to display multiple edges between nodes as they were always solid and thus less confusing. How best to display this new data? In the schema two other methods were outlined. The second of these was to have two edges with the call count next to them. The third was to display the most relevant edge, be it broken or solid. For the reasons outlined previously the third option was chosen. Thus for this algorithm there only ever exists one edge between the successor and predecessor in the case of single or multiple calls.

6.1.3 Comparison of outputs

Though all three algorithms arrive at call graphs it can been seen that the metadata extracted is quite different. It is interesting to note a number of important characteristics of these graphs that may not at first be apparent.

The first of these deals with recursion in code. In the naive and Steensgard algorithm any form of recursion results in what is, according to our schema, an infinite loop. Though this is useful in that it is possible to see where recursion occurs, if all recursion led to an infinite loop our programs would be quite useless! Secondly, if an infinite loop via recursion were to exist, this method of visualisation would be less useful. Due to Emami's algorithm being sensitive to the context of a function call it is possible to tell when a call might occur and it can be visualised accordingly. This method of visualisation is more useful straight away. If an infinite loop does exist, it will appear as a solid line. When a recursion has an escape case this is denoted as such by the use of a dashed-edge.

The second interesting finding is something that is very easy to miss or can easily be mistaken for an error in the extraction process. When describing the various algorithms one of the major factors was how the algorithms analysed individual functions. The two

methods for analysing functions were either inter-procedural, allowing sequential analysis or intra-procedural, which work in a similar fashion to an interpreter. Though it is easy to see the impact of these different methods on the extraction process; seeing the difference in graph output is a touch more complicated. To understand the difference, remember that to perform inter-procedural analysis every function in the application is extracted and then parsed. Instead using intra-procedural analysis every function call present in the call chains emanating from the arbitrary start function (main usually) is passed... This, as you may have guessed, means when using inter-procedural analysis all dead function blocks [1] are extracted and graphed. In intra-procedural analysis, these are not extracted and therefore not graphed. As an aside this *problem* can be solved by running the extraction process on every function in the application then grouping the results. See section 9.2.4 (page 93) for further details and discussion.

6.2 Force-Directed Graphs

The static graphs show a clear hierarchy of the program but are less useful when wanting to analyse specific call chains. With so much data being displayed on the screen it is harder to concentrate on one specific set of functions. Another problem is when there is too much data displayed it is hard to make sense out of all the edges and nodes. This section discusses a novel use of force-directed graphs for simplifying the graphs. See section 7.3.3 (page 68) for details of JSVIZ modifications.

6.2.1 Visualisation

How do these graphs look in comparison to the static graphs? As they are also a directed graph an attempt to follow the schema is made. A number of minor simplifications are required to facilitate their use. The nodes are still present, though instead of being in an ellipse they are designated with a dragging image and the name of the function. The edges are still drawn between nodes in the same way, only the lines are now only solid. Though these simplifications exist, the graph is augmented with other additions. Firstly the nodes in a graph can be moved. Corresponding edges and nodes are affected by this movement, this method of graph visualisation is known as a force directed graph, see section 2.7.4 (page 33). The second addition is that selecting a node will present a new view, the caller-callee view, which provides information directly relevant to the selected node. See figure 6.1

[1] Functions or sets of functions which are never used in a normal run of a program

(page 61) for an example.

6.2.2 Hierarchical View

The hierarchical view is very similar to the view displayed by the other static graphs view. This view presents a dynamic view of the entire program. As such it suffers from the same deficiencies as the static view, that of information overload. Due to the nature of force-directed graphs, this problem is further compounded leading to a confusing output at times. Fortunately this view does have one advantage, it presents a simple way to access the Caller-Callee View for every function. It also easily illustrates blocks of code where it would be prudent to obtain a better understanding of the code and call chains. See figure 6.1 (page 61)

6.2.3 Caller-Callee View

The caller-callee view takes a different angle to previous graphing methods. The other graphs attempted to provide an overview of all the functions that were analysed. At times it would be useful to display specific call chains in the code. This view works by allowing a view of these chains. Instead of displaying an arbitrary length chain the graph displays information specific to a selected function. The idea is based around a pivot function. The pivot is selected from the functions in the program. All functions that call the pivot and all functions that the pivot call are now added. This view makes it possible to walk forward in the chain, deeper into the function calls. It also gives the ability to work backwards; invariably until the root function *main* is reached. Figure 6.2 (page 62) shows two examples of the caller-callee view.

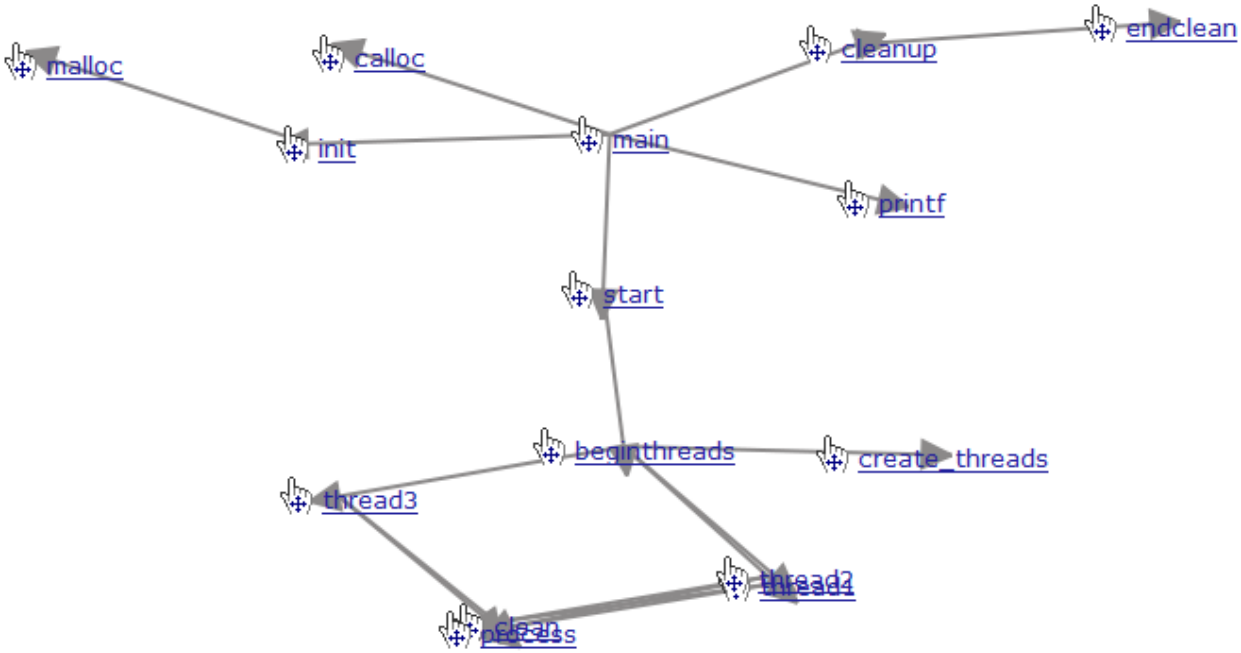
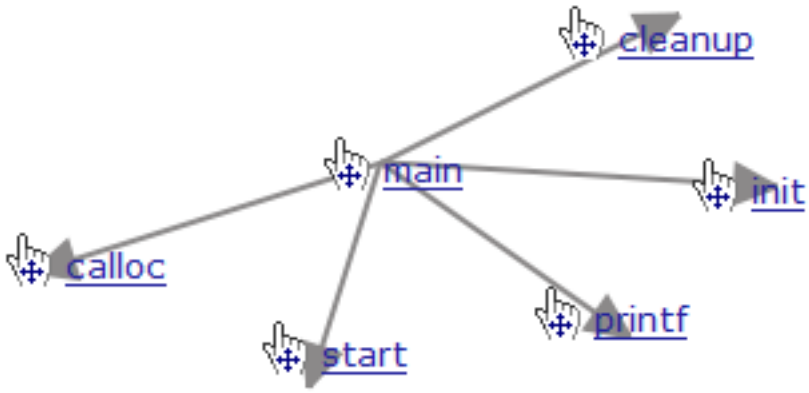
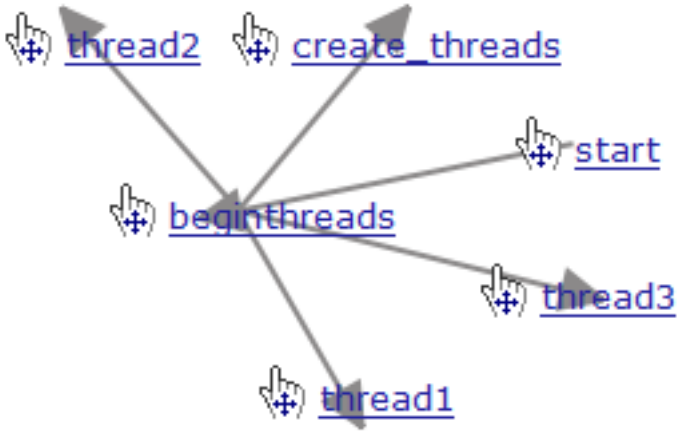


Figure 6.1: Hierarchical View



(a) View from Main



(b) View from beginthreads

Figure 6.2: Caller-Callee View

Chapter 7

Implementation Details

7.1 Requirements

In creation of the initial project proposal two main areas of research into call graphing became apparent. The first area dealt with the extraction of an abstract data type that represented the call chains in a program. The second area dealt with the visual representation of this data type. The initial requirements, appendix A.1 (page 107), were drafted in such a manner as to provide a flexible framework for work into either or both of these areas. The rest of this section should help to justify and illustrate the requirements that were outlined.

Call graphing in C is a well known area amongst compiler designers [1], those who have an interest in code optimisation and debugging. This has resulted in a large amount of research into the specific techniques of call graph extraction. Quite often the call graph will become part of a wider development tool chain [Fou07]. As a corollary to this; the users of such call graph extractors tend to be those with a good understanding of code, and programming paradigms. It follows that most would be familiar with such techniques as compiling a program from source code, in particular C code. Noting this, people who debug code, one of the primary audiences, may not be familiar with the various stages of compilation [2] thus the application should be usable by those people who are unfamiliar with such advanced, low-level problems. This leads to an impasse; how can an application be constructed that is both powerful and flexible enough to please both users?

[1] A clash graph can be constructed using similar techniques for pointer analysis. This is used to create the liveness set of a given function

[2] Lexical processing, semantic analysis and code generation

One of the most popular tool chains is the GNU tool chain [3] Thus most developers, at some point in their career would have come across these methods. Thus it is fair to assume that a call graph extractor that used a similar design ethos to these tools would be usable, and recognizable to most computer scientists. Thus a command line tool that worked in a similar method to GCC would meet these prerequisites. Continuing in this vain, most users of such a system would be familiar with passing command line arguments to an application. These arguments would control the extraction of the call graph.

Users of such a system will often be running a desktop machine with either Microsoft Windows or LINUX operating systems. Though the input code to the application should be treated as if it were machine independent. To facilitate users being on either of the various operating systems, the application should be developed in a portable, flexible language that can be compiled on many platforms. The application should also be developed such that it meets the requirements of the language standard. Another factor to take into account when deciding on an implementation language is the time it takes to extract the data from the input. In the implemented algorithms this ranges from O(n) to O(n^2) and for applications of a large size can be very computationally expensive. Thus a good implementation language choice would be the C compiler. In particular, the GCC compiler is available for both operating systems and is relatively standards compliant [4]. The requirements issue is how will the initial compilation stages be performed? The ubiquitous, LEX and YACC [Hil75, Joh79] are widely available, with versions available for both platforms. LEX provides a structured application to generate tokens based on lexical analysis. YACC provides semantic analysis to generate an abstract representation of the initial source code. As this project is not concerned with writing an ANSI C semantic checker, available LEX and YACC files can be utilised and modified to create this stage.

7.2 Design

The solution for the call graph extractor is divided into two main sections.

7.2.1 Graph Extraction

The first part of the application is a common infrastructure that can be used for the analysis techniques. This infrastructure at the lowest level take the form of a lexer and

[3] Make, GCC, Binutils, GDB, as well as a number of others.

[4] GCC has a native binary for windows and can also be run under cygwin.

parser. Output from this stage of the program will take the form of an Abstract Syntax Tree (AST). This provides a generic representation of the code into a standard format. The algorithms detailed in previous sections will use the tree as their basis for analysing the code, as such, the tree must be flexible enough to represent the various language constructs available in the C programming language.

The second part of the graph extractor is the implementation of the algorithms themselves. When invoking the application a command line switch is given that determines the algorithm to use. The algorithms parse the AST then are run against this. The completed graph extraction is output in an appropriate format to the standard out, as is usual with GNU command line tools. This output can then be processed by another application to aid in the creation of the visual representation of the graph.

7.2.2 Graph Visualisation

Once the call graph has been extracted from the source code it should be stored in an easily accessible format. An interface to convert the algorithm output to SQL was created. This allowed the output to automatically populate tables in a database. These tables correctly stored any associated metadata from the extraction.

To extract the correct data to create the graph images relied on various SQL statements. These were constructed in such a way as to extract all metadata independent of extraction method. These could then be displayed to the user. The list of extracted programs should be displayed on a web page allowing the user to dynamically view the extracted graphs.

The respective code for generating force-directed or static graphs [5] can be dynamically generated from the extracted database records.

7.3 Concepts

7.3.1 Union Find Data Structure

The Steensgard algorithm relies heavily on set operations, namely the union of two disjoint sets. The Union-Find data structure presents a constant time algorithm for performing the union operation.

[5] Javascript and DOT respectively

Algorithm

This algorithm starts with the problem of taking one type T_1 and another type T_2 and merging them together.

$$T_1 \bigcup T_2 \equiv T_3. \tag{7.1}$$

This new set contains the union of all these points. If one were implementing a naive algorithm for this operation the simplest solution would be to iterate through each point adding it to the new set. This would require 0(n) to union the two sets. The problem with this solution is that as sets get larger it becomes more time consuming to merge them together. A solution to this problem is the Union-Find data structure. The algorithm is based around two main functions, figure 7.1 (page 67), Figure 7.2 (page 67) and a data structure.

Given two sets, $s_1 \bigcup s_2$, the algorithm is as follows:

- Find the node s_1
  - Traverse the parent set s_1 until at node n_y which is at the head of the set
  - Repeat for s_2.
- Both heads of set s_1 and s_2 are known. Perform the following operation on these heads:
- if $s_1 = s_2$
  - In this case the sets are the same
  - Do not perform any operation
- else if s_1 != s_2
  - Set the head pointer of s_1 to s_2
  - This operation is the same as performing a union on the two sets

Complexity

The operation for joining two sets is a single operation resulting in a constant time complexity of O(1). It is worth noting that a penalty of O(m)[6] is incurred in finding the roots of the node. Though as the size of the individual sets tend to be significantly less than the total amount of nodes |m| < |n| this operation is quicker than the naive union algorithm. In particular it is quicker with respect to the actual runtime complexity; this is due to no memory needing to be copied [KST97].

```
1  FindSet( X )
2  {
3      while( X != root )
4      {
5          X = parent( X )
6      }
7      return X
8  }
```

Figure 7.1: Pseudo Code for the Find function

```
1  Union( X , Y ) /* X and Y are heads*/
2  {
3      if( X != Y )
4      {
5          Y->parent = X
6      }
7  }
```

Figure 7.2: Pseudo Code for the Union function

7.3.2 Emami's Algorithm: Mapping Parameters

As can be seen from Figure 7.3 (68) the first stage in the algorithm is to get the caller and callee function. This gets the current caller function which has the correct points-to set contained inside it. The callee function returned is a function which has only the blank function, and a list of variables contained in the function defined. This function is then copied, leaving the original clean version of the function untouched. The next stage is to get the actual parameters from the caller function and map them to the formal parameters in the callee function. This is only performed for variables of pointer type. Thus the points-to

[6] Where m is the size of the set being searched, this can be reduced using data structures such as a hash table

set of the callee function contains any pointers that were passed to it through the function call. Now that the callee function is correctly setup it is added to the list of functions that are called by the caller function. Now that the function is correctly setup and initialised it is passed the the main Emami algorithm and processed. Once the callee function and any subsequent calls are complete, the algorithm returns to the original caller [7].

```
1  calculateCall(AbstractSyntaxTree ast, bool conditional)
2  {
3      Function callee = getFunction( ast->FUNCTION_CALLED )
4      Function caller = ast->FUNCTION
5
6      /* Get a blank sheet of this function (I.e. Vars not initialised) */
7      Function newCallee = COPY( callee )
8
9      /* Set the new function up */
10     mapActualArguments( caller , newCallee )
11     addChild( caller->children , newCallee )
12
13     /* Parse the new function */
14     extractCalls_Emami( newCallee )
15
16     remapReturnValues( caller , callee )
17 }
```

Figure 7.3: Calculate functions calls in Emami algorithm

7.3.3 Modification of JSVIZ

As detailed in the literature review, section 2.7.4 (page 33) JSVIZ provides a force directed graph. These graphs are a nice way of dynamically displaying data using javascript inside of a browser. Unfortunately JSVIZ does not support the schema defined to show flow between nodes. JSVIZ even lacks the capability to create direct graphs which require arrows to indicate edge direction. A solution to this problem was to modify the JSVIZ source code to support direction.

Adding Arrows

Currently the edge between two nodes are defined by the two points in the xy-plane. A line is drawn between these two points where the first point (p_1) is the predecessor and

[7] Though this concept may at first appear to be rather complex! Seeming to require a large amount of structure manipulation, adding a lot of complexity to the algorithm, it may aid the reader to think of it in terms of an interpreter.

the second point (p_2) is the successor. With this information it is possible to devise a method for drawing an arrow between the points. If an arrow is drawn directly at point p_2 with these point being the base of the arrow it would be incorrectly oriented, see figure 7.4 (page 69). To solve this problem the first step is to translate the points so that p_2 lies on the origin. An arrow is now created at this point. Now that the arrow has been placed on the origin a right angled triangle with p_1 to p_2 as the hypotenuse. This triangle can be created by taking the Y co-ordinate of p_1 and the X co-ordinate of p_2. It is now possible to find out the angle which the triangle needs to be rotated by to correctly indicate direction, see figure 7.5 (page 70). The arrows points can now be rotated such that they are correct. Once the arrow is created the points and the arrow are translated back to their original positions.

(a) Incorrectly Oriented (b) Correctly Oriented

Figure 7.4: Arrow orientation on edge

Issues with Force Directed Graphs

The solution for adding arrows is correct in theory, but there is one problem with force directed graphs. As noted in the literature review the nodes shift moving the edges between them. This results in the arrows needing to be redraw every time a node moves. As javascript is an interpreted language running in a web browser these computationally expensive operations are detrimental to performance. To solve this problem one must realise that the arrow can only be rotated around 360^o. When JSVIZ initially loads an arrow is created for each angle and then cached. When a node moves, instead of calculating the exact rotation an approximation is taken from the cache. This caching results in many fewer mathematical operations needing to take place in the lifetime of the graph. This significantly reduces the operations in the long run and improves the drawing speed of Direct JSVIZ.

Summary

In summary the modification of JSVIZ to support arrows succeeded. This method provided an interesting and novel means of visualising the call graph. Unfortunately due to time constraints it was infeasible to implement all the schema defined in section 5.1 (page 53). The two biggest issues with not being able to implement the entire schema was that of showing recursion. In the static graphs it is easy to see where recursion occurs, this is denoted by an edge an arrow that start and end at the same node. Edges in JSVIZ are always a 2D point-to-point line, thus creating a self-referncing node would equate to a single point and an arrow. Thus the decision was taken to not show recursion in JSVIZ graphs. A modification of the drawing library to support curved lines would solve this. The problem of differentiating *may* or *will* edges is also an issue, all the JSVIZ edges are solid. Supporting variations on edges is feasible by supporting a variable field where the

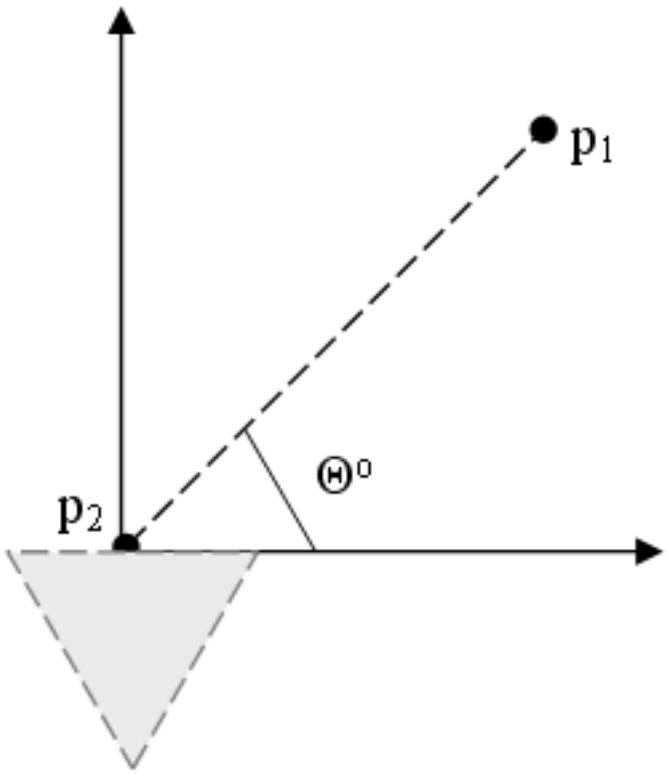


Figure 7.5: Rotation of triangle by θ

lines are rendered as dashed. As there is support for this in the SVG [8] standard this could be added relatively easily. The final problem with the JSVIZ implementation is inherent in the way force directed graphs work. As each node has a weight and attraction when too many of these nodes are in the system the physics falls apart. What is left is a confusing graph which is next to useless, see figure 7.6 (page 72). This problem is a well known issue with force-directed graphs, and as such is an open research issue [BHR95].

[8] Nodes in JSVIZ are SVG graphics.

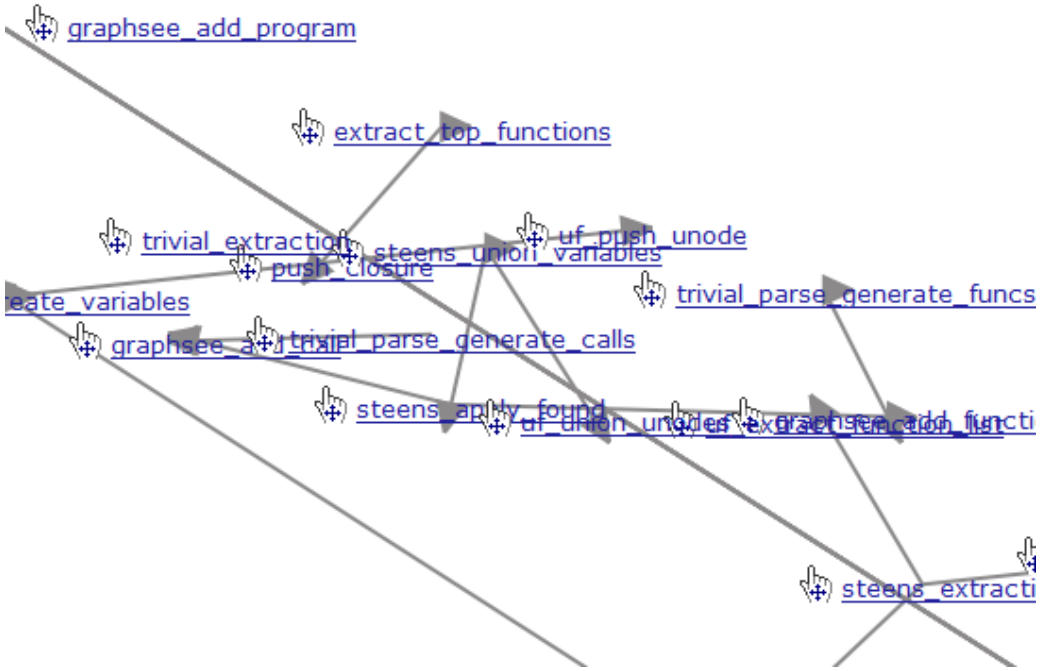


Figure 7.6: Force-directed graphs breaking when displaying too many nodes

Chapter 8

Experimentation

8.1 Introduction

Though the above section outlines the difference in implementation of the algorithms, it is necessary to illustrate these with experiments. The following is a selection of tests to be carried out on the algorithms. The justification and expected output for each algorithm is given.

The format for the each experiment output is:
algorithm type (input) ⇒ { output }
Function pointers have the syntax: **ptr_<name>**
Functions have the syntax: **func<name>**

$\phi \Rightarrow$ Empty Set
$\Delta \Rightarrow$ Input
$\lambda \Rightarrow$ All Algorithms
$\omega \Rightarrow$ May Call
$\Omega \Rightarrow$ Will Call
N ⇒ Naive Algorithm
S ⇒ Steensgard based Algorithm
E ⇒ Emami based Algorithm

Table 8.1: Key for Experiment Output

Note: If ω or Ω is not specified Ω is implied.

8.2 Simple Experiments

8.2.1 Experiment 1 - Empty Function Body

This experiment is a base test without any function calls, figure 8.1 (page 74). Thus the total set of calls for every algorithm should be the empty set.

Function	Output
Main	$\lambda(\ \Delta\ ) \Rightarrow \{\ \phi\ \}$

Table 8.2: Predicted experiment output of an empty function body

```
1  int main()
2  {
3    return 13;
4  }
```

Figure 8.1: Code for experiment 1

8.2.2 Experiment 2 - Trivial Function Call

This test contains a single function call in it, figure 8.1 (page 74). The call is located in the body of the main function. The called function has no calls of its own.

Function	Output
Main	$\lambda(\ \Delta\ ) \Rightarrow \{\ \text{funcZ}\ \}$
funcZ	$\lambda(\ \Delta\ ) \Rightarrow \{\ \phi\ \}$

Table 8.3: Predicted experiment output of an trivial function call

```
1  int funcZ()
2  {
3    return 12;
4  }
5
6  int main()
7  {
8    funcZ();
9    return 13;
10 }
```

Figure 8.2: Code for experiment 2

8.2.3 Experiment 3 - Recursion

This experiment gets a little more complicated as the problem of complexity is added into the equation. All algorithms should handle the recursion correctly.

Function	Output
Main	$\lambda(\ \Delta\ ) \Rightarrow \{$ funcZ $\}$
funcZ	$\lambda(\ \Delta\ ) \Rightarrow \{$ funcX $\}$
funcX	$\lambda(\ \Delta\ ) \Rightarrow \{$ funcZ $\}$

Table 8.4: Predicted experiment output of recursive functions

8.2.4 Experiment 4 - Conditionals

This experiment is the first where the algorithms should return a different result. The flow control IF-Statement is used. This should allow all algorithms which are sensitive to the context in which a call is made to return a possible call set.

Function	Output
Main	$N(\ \Delta\ ) \Rightarrow \{$ funcZ $\}$ $S(\ \Delta\ ) \Rightarrow \{$ funcZ $\}$ $E(\ \Delta\ ) \Rightarrow \Omega\{\phi\} \mid \omega\{$ funcZ $\}$
funcX	$\lambda(\ \Delta\ ) \Rightarrow \phi$

Table 8.5: Predicted experiment output when adding a conditional

8.3 Complex Experiments

8.3.1 Experiment 5 - Function Pointer

This experiment introduces the problem of function pointers to the equation, figure 8.5 (page 79). This new type should introduce a number of problems with the naive algorithm. The other algorithms should be able to handle the pointer correctly. As you will see the naive algorithm tries to make a call to a function that does not exist.

```
1  int main()
2  {
3          int (*ptr_fX)();
4
5          ptr_fX = &funcX;
6
7          ptr_fX();
8  }
```

Figure 8.3: Code for experiment 5

Function	Output
Main	N(Δ) ⇒ { ptr_fX } S(Δ) ⇒ { funcX } E(Δ) ⇒ { funcX }
funcX	λ(Δ) ⇒ ϕ

Table 8.6: Predicted experiment output using functions pointers

8.3.2 Experiment 6 - Function Pointer & Direct Call

This experiment continues in a similar vain to the previous one. Instead of just having a function pointer call, a normal call is included as well. This shows that the naive algorithm returns a partially correct result but still contains a call to a non-existent function.

Function	Output
Main	N(Δ) ⇒ { ptr_fX, funcX } S(Δ) ⇒ { funcX } E(Δ) ⇒ { funcX }
funcX	λ(Δ) ⇒ ϕ

Table 8.7: Predicted experiment output of direct & function pointer calls

8.3.3 Experiment 7 - Flow sensitivity

This experiment shows the flow sensitivity in the algorithm. In this experiment two function pointers are used. These are then assigned function names to point to. After this assignment an update is performed that updates the values stored it in the second pointer. In the Emami algorithm this gives a hard update and removes the function that it originally pointed to. The important thing to remember is that the value in the first pointer is not changed to what was stored in the second pointer. In the Steensgard algorithm, which is not flow sensitive, the first pointer and the second pointer contain the same values. Thus

when a call is made both functions *may* be called. The naive algorithm misidentifies both function pointer declarations as calls.

Function	Output
Main	N(Δ) $\Rightarrow$ { ptr_fX, ptr_var2 } S(Δ) $\Rightarrow$ { funcX , funcY } E(Δ) $\Rightarrow$ { funcX }
funcX	λ(Δ) $\Rightarrow$ ϕ

Table 8.8: Predicted experiment output with flow sensitivity

8.3.4 Experiment 8 - Function Pointer & Context Sensitivity

All three algorithms will return a different result set in this experiment. The flow control IF-Statement is used as well as function pointers. In theory the naive algorithm will ignore the flow statement and misinterpret the function pointer as a function. Steensgards algorithm will correctly identify the function pointer, but will ignore the IF-Statement adding the function into the definitely calls set. Emamis algorithm will correctly identify the algorithm and that it will not definitely get called as it resides in the conditional.

Function	Output
Main	N(Δ) $\Rightarrow$ { ptr_fX } S(Δ) $\Rightarrow$ { funcX } E(Δ) $\Rightarrow$ Ω{ ϕ } | ω{ funcX }
funcX	λ(Δ) $\Rightarrow$ ϕ

Table 8.9: Predicted experiment output of function pointers and context sensitivity

8.3.5 Experiment 9 - Strong Updates

All three algorithms will return a different result set in this experiment, figure 8.4 (page 78). In this experiment the pointer is assigned a function and then re-assigned to another function. None of the assignments occur in a conditional statement. The naive algorithm incorrectly identifies the function pointer as the name of the function being called. In the case of Steensgard's algorithm the pointer is correctly identified but the update is weak. As such, the algorithm interprets the call as one to both functions. Emamis algorithm first points to funcX and then when the second assignment to funcY occurs it is a strong update. This overwrites the previosuly pointed to function. When the pointer is called, the only function called is funcY.

```
1  int main()
2  {
3          int (*ptr_var) ();
4
5          ptr_var = &funcX;
6
7          ptr_var = &funcY;
8
9          ptr_var();
10 }
```

Figure 8.4: Code for experiment 9

Function	Output
Main	N(Δ) $\Rightarrow$ { ptr_var } S(Δ) $\Rightarrow$ { funcY, funcX } E(Δ) $\Rightarrow$ { funcY }
funcX	λ(Δ) $\Rightarrow$ ϕ

Table 8.10: Predicted experiment output with strong updates

8.3.6 Experiment 10 - Context Sensitivity & Conditonal Statement

The next experiment tests the context and handling of flow control of a piece code. The code works around the introduction of an IF-Statement into the experiment. The pointer is assigned a value both before and then inside the IF-Assignment.

Function	Output
Main	N(Δ) $\Rightarrow$ { ptr_var } S(Δ) $\Rightarrow$ { funcY, funcX } E(Δ) $\Rightarrow$ Ω{ funcY } | ω{ ϕ }
funcX	λ(Δ) $\Rightarrow$ ϕ
funcY	λ(Δ) $\Rightarrow$ ϕ

Table 8.11: Predicted experiment output with context sensitivity & conditional statements

8.3.7 Experiment 11 - Context & Flow Statement

This experiment has many parallels with previous experiments. The code works around the introduction of an IF-Statement into the experiment. The pointer is assigned a value both before and then inside the IF-Statement an assignment. The change is that after the IF-Statement another assignment takes place. This strong update means that it is definitely known what the pointer points-to at its invocation.

Function	Output
Main	N(Δ) $\Rightarrow$ { ptr_var } S(Δ) $\Rightarrow$ { funcY, funcX } E(Δ) $\Rightarrow$ Ω{ ϕ } | ω{ funcX, funcY }
funcX	λ(Δ) $\Rightarrow$ ϕ
funcY	λ(Δ) $\Rightarrow$ ϕ

Table 8.12: Predicted experiment output with context & flow statements

8.3.8 Experiment 12 - Passing Pointers as Parameters (1)

This experiment is quite different from previous. This shows the difference in results that are achieved by using intra-procedural analysis. A function pointer is passed to an argument and then called, figure **??** (page **??**). This results in the points-to set of the callee function being modified during different invocations.

```
1  int funcY(int *ptr_func)
2  {
3    ptr_func();
4    return 9;
5  }
6
7  int main()
8  {
9        int (*ptr_var) ();
10
11       ptr_var = &funcX;
12
13       funcY(ptr_var);
14
15       return 21;
16 }
```

Figure 8.5: Code for experiment 5

Function	Output
Main	N(Δ) $\Rightarrow$ { ptr_var } S(Δ) $\Rightarrow$ { funcY } E(Δ) $\Rightarrow$ { funcY }
funcX	λ(Δ) $\Rightarrow$ ϕ
funcY	N(Δ) $\Rightarrow$ { ptr_func } S(Δ) $\Rightarrow$ { ϕ } E(Δ) $\Rightarrow$ { funcX }

Table 8.13: Predicted experiment output passing pointers as parameters

8.3.9 Experiment 13 - Passing Pointers as Parameters (2)

This experiment also shows the difference in results that are achieved by using intra-procedural analysis. A function pointer is passed to an argument and then called. Inside the called function is a conditional statement that changes the assignment of the parameter. The naive algorithm incorrectly identifies the name of the pointer as the function being called. As Steensgard's algorithm is inter-procedural the pointer is not passed to the function. This results in the only value being assigned to the pointer is the one inside the body. In Emami's algorithm the pointer is assigned the function funcX before it is passed. After it is passed a weak update occurs with the function funcZ being assigned. When a call is made to the pointer Emami's algorithm identifies that the call *may* be to either funcX or funcZ.

Function	Output
Main	N(Δ) $\Rightarrow$ { ptr_var } S(Δ) $\Rightarrow$ { funcY } E(Δ) $\Rightarrow$ { funcY }
funcY	N(Δ) $\Rightarrow$ { ptr_func } S(Δ) $\Rightarrow$ { funcZ } E(Δ) $\Rightarrow$ { Ω { ϕ }$\mid$ ω{ funcZ, funcX }
funcX	λ(Δ) $\Rightarrow$ ϕ
funcZ	λ(Δ) $\Rightarrow$ ϕ

Table 8.14: Predicted experiment output with passsing pointers as parameters

8.3.10 Experiment 14 - Field Sensitivity - Pointer Array

This experiment brings deals with the case when there exists function pointer arrays in the application. The various indices in the array are assigned function values. A number of these are then invoked. The naive algorithm incorrectly identifies the array name as the function being called. The Emami's and Steensgard's algorithms cannot destinguish which indices are being called and therefore pressume that all have been called.

Function	Output
beginthreads	N(Δ) $\Rightarrow$ { createthreads, ar } S(Δ) $\Rightarrow$ { createthreads, thread1, thread2, thread3 } E(Δ) $\Rightarrow$ { createthreads, thread1, thread2, thread3 }

Table 8.15: Predicted experiment output with function pointer arrays

8.3.11 Experiment 15 - Definite vs. Indefinite Recursion

This experiment shows where there is recursion that is either definite of indefinite. This illustrates how the effect of a conditional statement can alter the output of the various algorithms. In the naive and Steensgard's algorithm the conditonal statement is ignored; resulting in a definite call from the function cleanup to itself. In Emami's algorthm it is correctly identified that cleanup *may* call itself. This results in the indefinite recursion.

Function	Output
cleanup	N(Δ) $\Rightarrow$ { empty, cleanup, endclean } S(Δ) $\Rightarrow$ { empty, cleanup,endclean } E(Δ) $\Rightarrow$ { Ω { empty, endclean }| ω{ cleanup }

Table 8.16: Predicted experiment output with definite and indefinite recursion

8.3.12 Experiment 16 - Dead Function Blocks

This experiment deals with the case when there are dead function blocks in the code. These functions are ones where they will not be directly called from the main branch of the application. As can be seen from the predicted output there are two lots directed graphs that would be extracted in the case of the Emami and Steensgard algorithm. As the Emami algorithm works only from the main function the second directed graph would not be extracted.

Function	Output
Main	λ(Δ) $\Rightarrow$ { start, clean }
Start	λ(Δ) $\Rightarrow$ { process }

Table 8.17: All algorithm predicted experiment output - non-dead function block

Function	Output
printResults	N(Δ) $\Rightarrow$ { printArray } S(Δ) $\Rightarrow$ { printArray }
printArray	N(Δ) $\Rightarrow$ { printLine, printf } S(Δ) $\Rightarrow$ { printLine, printf }

Table 8.18: The Naive and Steensgard algorithm predicted extra - dead function block

```
1  int main()
2  {
3      start();
4      clean();
5  }
6
7  int start()
8  {
9      process();
10 }
11
12 int printResults()
13 {
14     printArray();
15 }
16
17 int printArray()
18 {
19     printLine();
20     printf();
21 }
```

Figure 8.6: Code for experiment 16

Chapter 9

Results & Evaluation

The following chapter looks at the results of the experiments outlined in section 8 (page 73). The experiments are slanted towards testing specific cases against the algorithms. Fortunately these cases also provide interesting output to evaluate the various graphing mechanisms. Though there are many experiments only a few will be dealt with. Those that offer interesting results are detailed to illustrate concepts. An empirical listing of the algorithms results can be seen in Appendix A.7 (page 116). Finally an evaluation of the various methods is carried out.

9.1 Algorithm Results

The following evaluations looks to see if the results of the experiments was as is expected. Secondly the correctness[1] of the results is looked at.

9.1.1 Trivial Function Call

The simple piece of code in experiment 2 calls funcZ from main. The output for all three algorithms was the same; a single call from main to funcZ. These are the expected results and correct for all three. This piece of code, though not complex, illustrates that the various algorithms can in some situations have the same output.

[1] Does the output correctly reflect the structure of the code

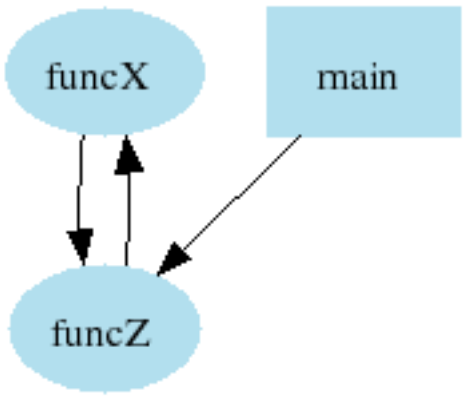


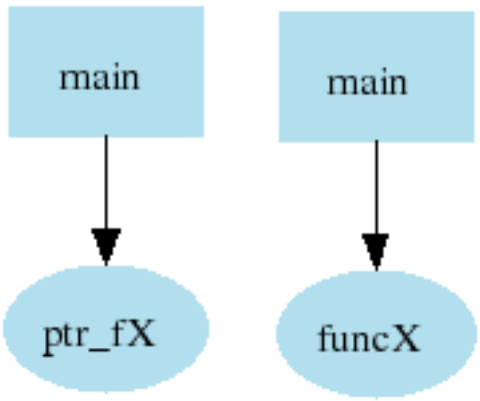
Figure 9.1: Graph of recursive function call for all algorithms

9.1.2 Recursion

Experiment three deals with the scenario where indirect recursion is present. As the naive and Steensgard algorithm independently analyse each function there is no danger of them not completing. In each case the function calls are correctly identified and the algorithm completes. In the case of the Emami algorithm it is conceivable that the algorithm may miss the recursion, or worse yet never finish analysing the code. In this case the indirect loop was correctly identified via checking the call chain and the recursion ended. It is interesting to note that once again each algorithm completed with the same output even though the analysis methods were quite different. See figure 9.1 (page 84).

9.1.3 Conditionals

As was seen from the Trivial Function Call experiment, all three algorithms correctly handled function calls. In this experiment a conditional statement has been added around the function call. Using the naive and Steensgard algorithm this statement was ignored. Both algorithms correctly found the function call and created the correct output. When the Emami algorithm was run against the code it correctly noted that a conditional was hit as such the function call falls into the *may* call set. All three algorithms go to the correct output.



(a) Naive Algorithm (b) Steensgard & Emami Algorithm

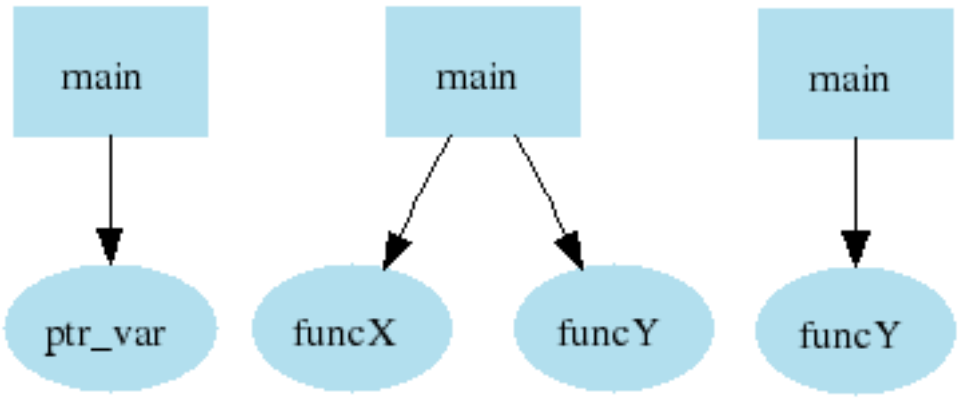
Figure 9.2: Graph of function pointers

9.1.4 Function Pointers

Experiment 5 added a new problem to the analysis stage, that of function pointers. In the code funcX (a function) was assigned to the pointer. Looking at the output Steensgard's and Emami's algorithm identified the variable as a pointer. These algorithms tracked the assignments arriving at a call to funcX. Both algorithms produce the expected and correct output. In the case of the naive algorithm this function pointer was not identified. Instead when the call is made via the pointer, it is assumed that the name of the pointer is the function being invoked. Thus the output now has incorrectly identified a call to ptr_fX and has no knowledge of any call to funcX. The output of the naive algorithm is as expected due to its lack of pointers, though the actual results are incorrect and can lead to confusion. See figure 9.2 (page 85).

9.1.5 Strong Updates

In experiment 9 the situation occurs where a pointer is assigned a function and then re-assigned to a new function. Following from the previous experiment the naive algorithm does not identify the function pointer and produces expected but incorrect results. Where the analysis gets more interesting is with Emami's and Steensgard's algorithm. As was outlined in the algorithm explanation the pointer analysis provided by Emami's algorithm is context and flow sensitive. Thus under the correct situations an update can either be strong or weak. In this experiment the update is a strong one, overwriting any previously



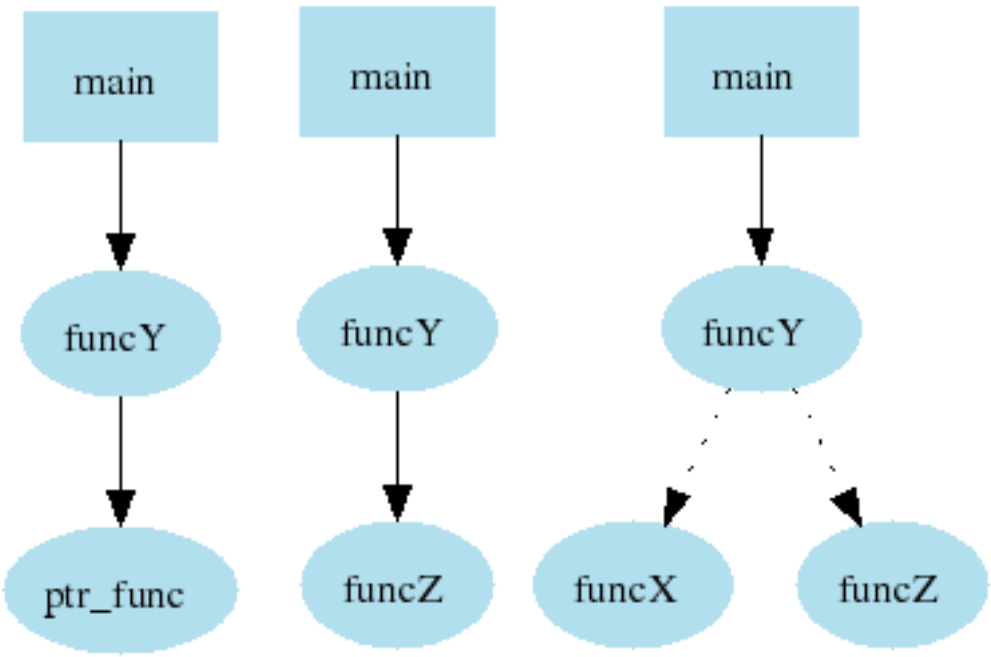
(a) Naive Algorithm (b) Steensgard Algorithm (c) Emami Algorithm

Figure 9.3: Graph of strong updates

held values. Thus when the function pointer is eventually invoked the only function pointer to is the last assigned one, funcY. Unlike Emami's algorithm Steensgard's is context and flow insensitive thus when a second value is assigned to the pointer it is added to the list of functions. when the pointer is invoked it is recognized as a call to funcX and funcY. Both algorithms produce the expected output. Steensgard's algorithm is incorrect as the call to funcY can never take place; Emami's algorithm is correct in that the only call will be to funcX. See figure 9.3 (page 86).

9.1.6 Passing Pointer as Parameters

Experiment 13 deals with passing function pointers as parameters. As has already been established the output for the naive algorithm is as expected but is incorrect. When looking at Emami's and Steensgard's algorithms a number of interesting results are found. Looking at Steensgard's algorithm, it is known that *main* invokes funcY. As the analysis is inter-procedural it is impossible to tell any effects that *main* may have on funcY. Inside of funcY the parameter is assigned to again inside a conditional. As the parameter did not have any pointers passed to it it now only contains a link to funcZ. Thus when the pointer is invoked a call to funcZ is noted. This result is as expected, though with the knowledge that another function was passed to funcY this output is not necessarily correct. Emami's algorithm performs intra-procedural analysis of this code and as such the effect of passing a pointer from *main* to funcY can be analysed. In the scenario the parameter begins by having the function funcX assigned to it (as passed in from main). In the body of the code a weak update takes place and as such the pointer now points to both funcX and funcZ. When the



(a) Naive Algorithm (b) Steensgard Algorithm (c) Emami Algorithm

Figure 9.4: Graph of passing pointers as parameters

pointer is invoked the algorithm detects that either funcX or funcY may be called. The output from Emami's algorithm is as expected and is correct. See figure 9.4 (page 87).

9.1.7 Function Pointer Array

One of the interesting cases noted in the algorithm design was the situation where arrays of function pointers exist. The naive algorithm once again fails producing the expected but incorrect result. Both Emami's and Steensgard's algorithms are field insensitive, treating arrays as a single variable. This is correctly detected by both algorithms. On invocation of the pointer all the functions {Thread1 | Thread2 | Thread3 { are called from beginthreads. The only difference is that Emami's algorithm recognises that only one of the three functions will actually be called and puts them into the *may* call set. In conclusion both algorithms produce the expected output. It could be argued that both of the algorithms are correct as the output correctly recognises all flows between functions. If a decision had to be made about which was more correct, Emami's output results in a better reflection in the logical nature of the code. See figures 9.5, 9.6 and 9.7.

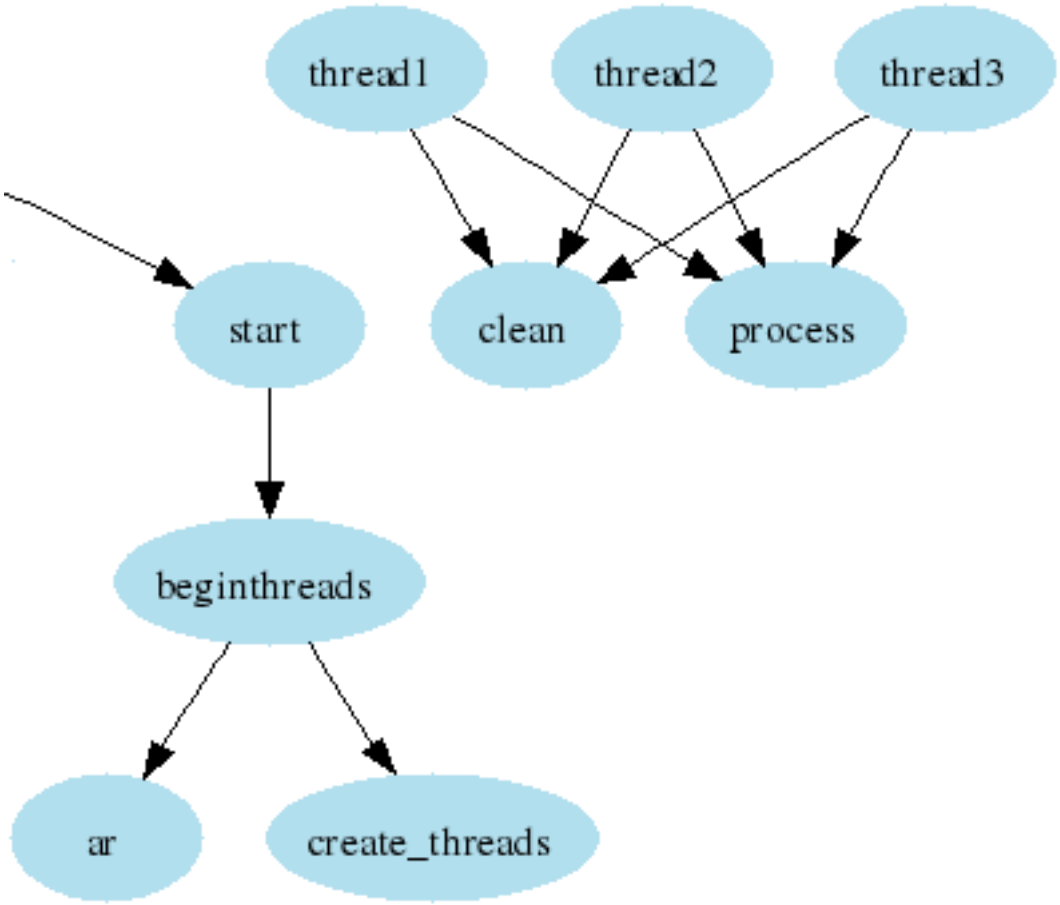


Figure 9.5: Graph of using function pointer arrays - Naive Algorithm

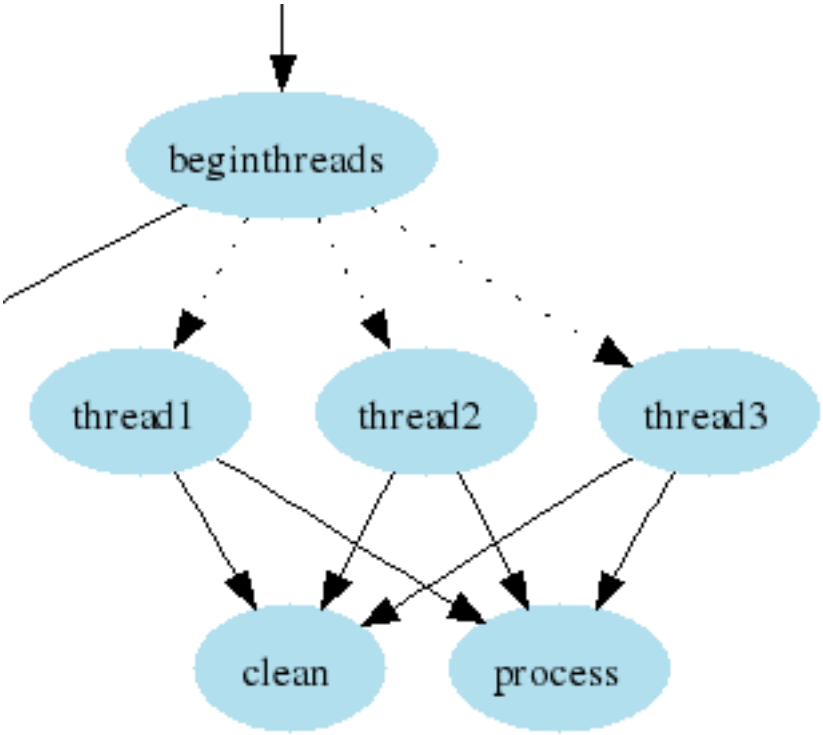


Figure 9.6: Graph of using function pointer arrays - Steensgard Algorithm

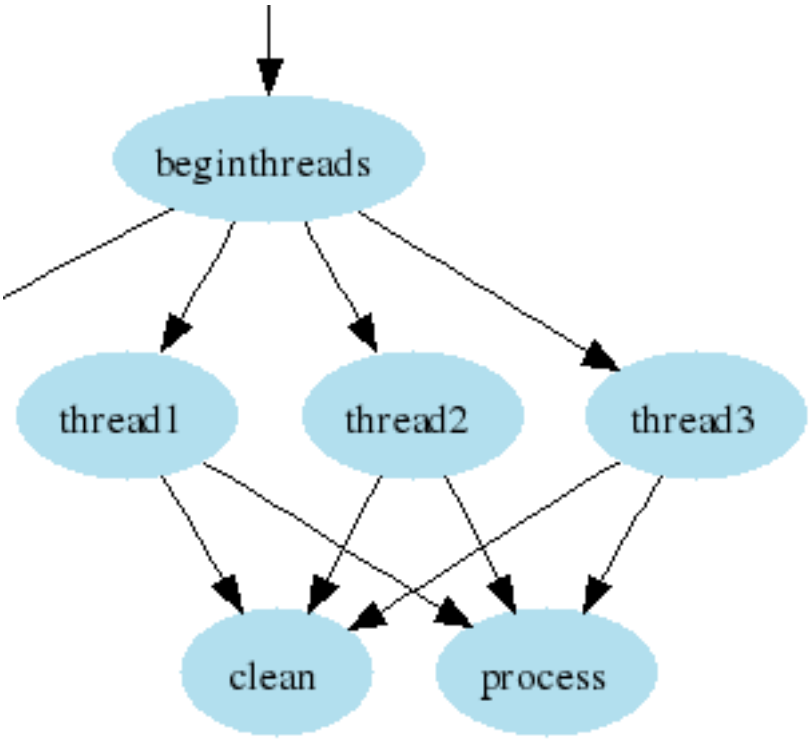


Figure 9.7: Graph of using function pointer arrays - Emami Algorithm

9.1.8 Inter-procedural vs. Intra-procedural

The final experiment produces some very interesting results which are very hard to justify in terms of correctness. Unlike the other experiments this has a section of function calls that are unreachable, as such are never called on any possible program run. The naive and Steensgard's algorithm result in the same output (note: function pointers are not used). When looking at the output it is both as expected and correct. The graph reflects what would happen if the program would be run [2]. These algorithms also correctly identify and output the pieces of code that are not in the main branch of the application, though this does not make the output more or less correct it does give more detail. Emami's algorithm is intra-procedural and works through the main call body of the program. The results from this algorithm are once again as expected and correct. This time though the extra uncalled function chains are not present. See section 9.2.4 (page 93) for further discussion and figure 9.11 (page 94).

9.2 Graph Results - Static Graphs

9.2.1 Function Call

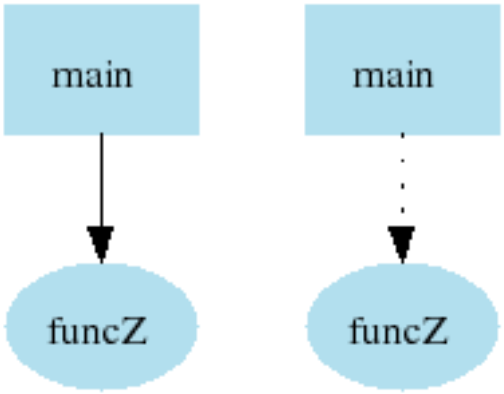
As noted in the algorithm results the three algorithms had the same intended output. Though the graphing algorithms are different it can be seen, figure 9.8 (page 92), the same results are produced. This case is interesting as it illustrates that once again, the different algorithms can create the output. You can also see that the main function is a rectangle shape whereas the other functions are ellipses. The graphs created correctly reflect the previously defined standards for graph generation.

9.2.2 Conditional Calls

In the algorithm results section it was noted that the output for experiment 4 by Emami's algorithm was different from the other two. As the call from *main* fell into the *may* call set it should be denoted with a broken edge. As can be seen in figure 9.9 (page 92) Emami's algorithm has a broken line between the calls whereas the algorithms have a solid line. Thus our graphing algorithm created the correct visualization of the experiment output. The graphs created from the algorithm output correctly reflect the previously define output.

[2] Note that C programs start from main

Figure 9.8: Graph of trivial function call



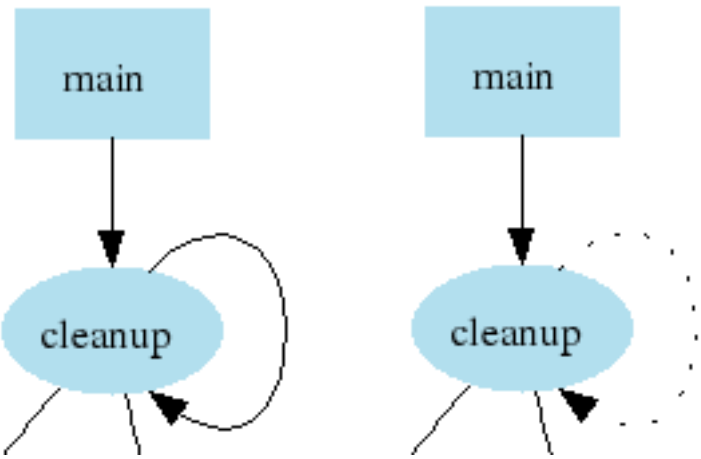
(a) Naive & Steensgard Algorithms (b) Emami Algorithm

Figure 9.9: Graph of conditional calls

The conditional calls are correctly identified in the graph using modification of the edge.

9.2.3 Definite vs. Indefinite Recursion

In experiment 15 there exists a direct recursive call. Such that the *cleanup* calls itself. As it would be a good idea that the code eventually ends there is an conditional statement that allows it to stop recurring at some point. As the naive and Steensgard's algorithms do not take into account conditional calls both of these are designated as a solid line in the graph. As Emami's detects that the call is inside a conditional it is shown with a broken line indicating it may get called. See figure 9.10 (page 93). The graphs created from the



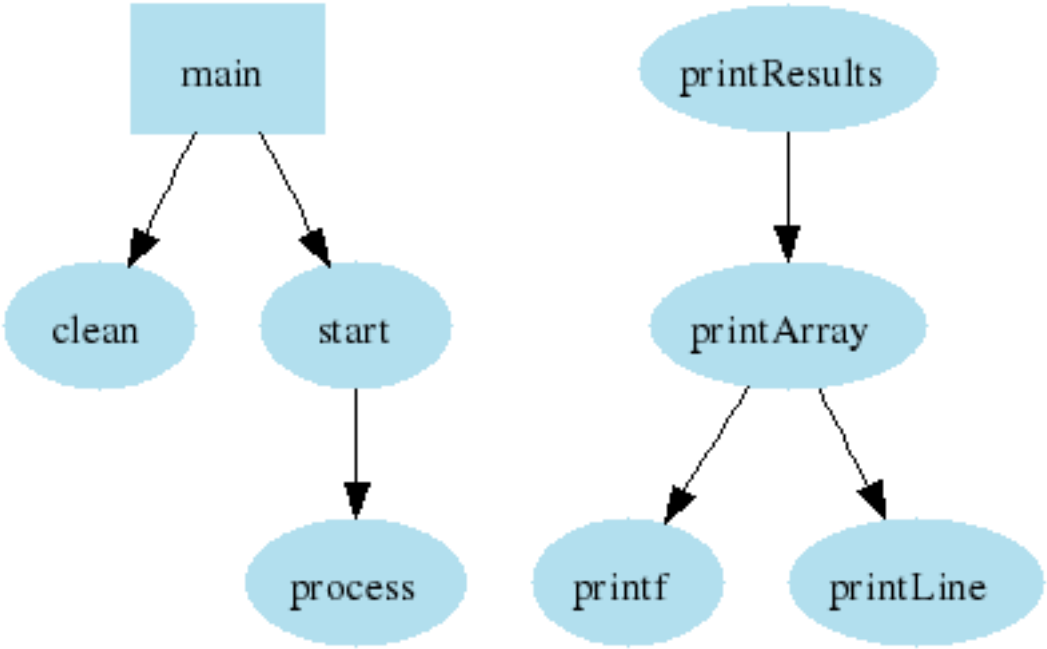
(a) Naive & Steensgard Algorithm (b) Emami Algorithm

Figure 9.10: Graph of various recursion types

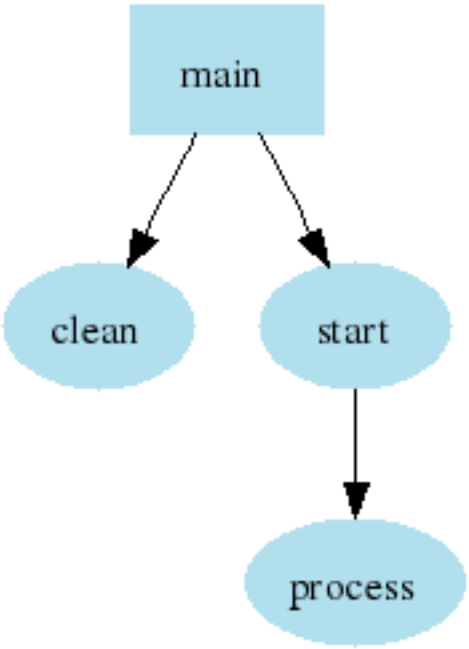
algorithm output correctly reflect the previously define output.

9.2.4 Dead Function Blocks

In the algorithmic results of experiment 17, Inter-procedural vs. Intra-procedural, the different types of analysis produce different results. The intra-procedural did not look at the uncalled chains whereas the inter-procedural did. As shown in section 9.1.8 (page 91) this results in dead function blocks being in the output. This experiment directly shows the visual results of this. In the graph for the naive and Steensgard algorithm two independent graphs can be seen. The one graph has *main* at its root, this is the main program path. The second graph indicates a separate call chain that will never be invoked, such as redundant code. The more complex Emami algorithm did not analyse these dead function blocks and the resulting graph reflects this. See figure 9.11 (page 94). The graphs created from the algorithm output correctly reflect the previously define output. The graphs show a distinct difference in the algorithms, both being correct, but one showing extra data.



(a) Naive & Steensgard Algorithms



(b) Emami Algorithm

Figure 9.11: Differences in graphs using Inter-procedural vs. Intra-procedural

9.3 Conclusions

9.3.1 Code Analysis

The experiments in this evaluation show an interesting trend. When analysing relatively simple code without too many complex data structures all three algorithms perform admirably, and in most cases get the same results. When looking at conditional statements Steensgard and the naive algorithm produce similar results with only the only Emami algorithm being more correct. As soon as the code gets more complex and the problem of function pointers is added the naive algorithm starts to break down. Without the capability to recognise pointers the results or the algorithm become very incorrect with inexistent functions being called and real functions being ignored. The two pointer based algorithms continue in a similar vein until presented with the possibility to perform strong updates. From here on the results of Emami's algorithm provide a better reflection of the code.

9.3.2 Function Call Analysis

Another conclusion that can be extracted from the experiments is that the method, intra-procedural vs. inter-procedural can have a large impact on the results of output. It is not clear from the experiments if one is better from the other. Inter-procedural analysis allows for the possibility of dead functions to be viewed and from the experiments gives a *good enough* approximation of the application. The intra-procedural analysis gives a more correct representation of the algorithm, but suffers from not presenting as good a structural overview of the application, one of the aims when setting out on this dissertation.

Chapter 10

Conclusion

10.1 Appraisal

The initial goal of this dissertation was to look into the process of creating a structural overview of a C program. From the literature review it became apparent that the process of call graph creation can be logically split into two stages; extracting then visualising the function calls. In both of these stages the dissertation has been a success.

The first aim was to produce a call graph extractor for C. When looking at this problem a number of solutions for understanding the structure of the program were avaliable. Following from previous work in creating a MIPS code generator it seemed appropriate to develop an abstract syntax tree (AST). This AST could be used as a framework to develop the extraction algorithms on. Freely avaliable FLEX and BISON definitions were modified to create the AST. These definitions were complete with respect to ANSI C. The only issue was that ANSI C has a very complicated syntax and semantics; to expedite the creation of the AST a number of simplifications were made. For instance; when developing the framework it made sense to ignore syntax such as decrement and increment operators [1] as function calls generally do not involve these operations. Though this seemed like an appropriate decision at the time, it was difficult to forsee all the scenarios needed to test the algorithms. The end result was that the FLEX and BISON files were modified numerous times during the course of the dissertation.

An appropriate framework had now been created to manipulate the source code. The

[1] ++ & − −

first aim from this point was to develop a naive algorithm to extract call graphs. When originally implementing the algorithm a choice of whether to process each function independently or start at some arbitary function, usually main, was needed. Originally it seemed that the naive algorithm would just process the function from main. Once the algorithm had been implemented it was tested on a number of the experiments. It soon became apparent that this algorithm would get stuck on recursion and infinitly loop until running out of memory. It would be possible to stop this recursion by keeping a list of functions in the current call chain. This solution did not seem appropriate as the algorithm would diverge from its original aim of being naive. Instead the algorithm was changed to process each function independently. In retrospect this was a good decision as it allowed the trivial algorithm to be a base for Steensgard's algorithm. Ultimately, the naive algorithm still failed when encountering function pointers. As such an improvement was needed.

As discussed in the literature review, Steensgard's code analysis technique allowed for approximating the call set of function pointers. Steensgard's algorithm was based on inter-procedural analysis, the same used as the naive algorithm. It was possible to use the knowledge gained from implementing the naive algorithm in this solution. One of the biggest challenges when using the Steensgard algorithm was that of understanding the abstract types. These types made the algorithm type safe and provided a mathematical representation for the operations. To implement this required the use of an unfamiliar data structure: the Union-Find data structure. After researching the data structure it was possible to implement a generic version; this was then used in the algorithm. After the initial challenges of understanding the abstract types, the implementation of this algorithm went smoothly.

For the final extraction method it seemed appropriate to use a different technique, that of intra-procedural analysis. In the literature review one method for approximating the call set of function pointers used intra-procedural analysis: Emami's algorithm. This technique also provided a method of avoiding the recursion problem that was encountered during the creation of the original naive algorithm. Unlike the Steensgard based algorithm this implementation proved less than simple. Two main issues occured when creating the algorithm. The first was that the concept of the algorithm was at a tangent to previous algorithms. Whereas the method for analysing each function had been the same, iteration over each, a more complicated *interpreted* technique was now needed. The second issue was that the algorithm added a lot of extra features. Namely: recognising weak and strong updates, passing pointer parameters, and taking into account conditional statements. This resulted in a much more complex implementation.

Using inter-procedural analysis the parameters of function calls could be ignored. In Emami's algorithm the actual parameters needed to be mapped to formal parameters. In the final implementation the arity of each function was restricted to a single parameter. Though this restraint is currently in place the algorithm could easily be modified to support multiple parameters. Emami also provides guidance for returning pointers from a function. The final implementation did not support returning pointers. In retrospect this would have been an good idea. It would have been able to expand our analysis to take into account this problem. That said, the ability to return function pointers could be added without too much pain. When a callee function returns the *will* and *may* call sets designated in the return statement could be returned. These could then be added to the sets of any pointer being assigned to in the caller.

The second complication was the problem of conditional statements and weak/strong updates. This seemed at first to be a very complicated issue. In actuality, when implementing this functionality it was possible to set a flag that indicated if the current statement was in a conditional. Depending on the value of the flag, assignments and calls could be added to the *will* or *may* call sets.

Finally, being able to compare and contrast three very different algorithms resulted in some interesting findings. The most important of these, not discussed in any of the associated literature, was that of dead function blocks. To recap, dead function blocks are uncalled functions or sets of functions. In many ways it is not surprising that this may have been missed; the author originally thought the variations in graph were due to a fault in the implementation rather than a side-effect of the analysis. The ability to locate unused blocks of functions can result in the optimisation of the compilation stage and be useful in the debugging process of program development (See Future Work).

The second aim of the dissertation was to visualise the extracted call graphs from the algorithms. The problems of representing calls was discussed. It soon became apparent that a schema for describing the output was needed. This formed a standard for the various visualization techniques; uses of this schema with respect to the extraction algorithms was discussed. This provided two variations for displaying the extracted call graphs. The abstract representation of the call graph had many similiarties with a directed graph. As mentioned previosuly a trivial mapping between the call graph and a directed graph was made: nodes corresponding to functions and directed edges to function calls. In the literature review various methods for creating graphs were discussed. One implementation, GraphViz, provided static graphs that were created using a language called DOT. To create these graphs the abstract representation of the call graph was turned into the DOT

language; graphs were then dynamically created. Though this proved easier than the extraction process a number of difficulties were encountered. The most obtrusive of these occured when the number of nodes in the graph increased. To combat this a scale variable was placed on the graph size. This scale was based on the number of nodes in the system. The final result of this graphing technique was aesthetically pleasing and provided a good structural overview of the code.

The second technique involved the use of dynamic force-directed graphs. Though force-directed graphs are a well known idea, a novel implementation was used. The concept of being able to view the construction and final result of the forces on the nodes in real-time, all done in a web browser. The solution was based on the JSVIZ graphing library. This library was modified to support directed edges. The modification encompassed the use of a number of graphics techniques to dynamically add and rotate an arrow located on vertices. Being able to use dynamic force-directed graphs lent itself to two different views. The first was a structural overview; similar to that provided by the static graphs. The second aimed to present a simpler view of the data and to allow the user to select which call chains to view. This was based on caller-callee relationships. Finally it was found that though force-directed graphs excel at showing caller-callee relationships, they are unsuitable with a large number of nodes. The use of this graphing technique in a web-browser exacerbated the problem. The interactions between nodes resulted in unpredictable actions and the nodes never reaching a reasonable equilibrium.

10.2 Future Work

As the project was divided up into two main section it seems pertinent to discuss future work in a similar fashion.

10.2.1 Extraction

From a research perspective being able to use the contrasting results between intra-procedural and inter-procedural analysis could be very interesting. Being able to eliminate dead blocks of code/functions could result in smaller applications with less code to understand and maintain. This analysis could conceivably speed up the compilation process by stripping unused functions. In an application containing millions of lines of code this technique could have a profound effect. Though it stands to reason that this would have a minimal effect

on the runtime of an application. The ignored code would have never been run in the first place.

A second use of this analysis, which could lead to improved performance concerns finding bottlenecks in an application. Currently programs such as DTRACE or MemSpy [MMG06, MGA92] use kernel level monitoring to detect invocations of functions. Analysis such as these require the application to be running and access to the kernel to detect these bottleneck areas. Quite often developers will not know the source of these bottlenecks [MGA92] and as such diagnostic tools are the only method of finding the problem. Being able to quickly see which functions are called by many other functions would be a good starting point for debugging. Functions which are critical to the application can be optimised or restructured to be more efficient. These critical functions, areas that may cause bottlenecks, can easily be detected. During the graph rendering process nodes could vary in colour depending on how many functions invoke them to further aid this process.

Looking at the analysis of code one of the biggest limitations is what to do when library functions are called. As is generally the case with library calls the code is compiled; straight source code analysis is impossible. One could argue that the analysis should be carried out on the library code and as such it is correct that this is ignored. Consider the case when a function returns a function pointer, a popular paradigm in C (see section 2.3.2 (page 20)) currently there is no way to tell what function was returned. Furthermore, taking the worst case results in the conclusion that the pointer could point to any function [2]. One solution to this problem would be the use of code de-compilation. The called function could be decompiled and then the code analysed as per-usual.

From an implementation perspective the extraction framework presents a good base for building the algorithms. At this stage the framework is not a complete reflection of the ANSI C standard or even C as it is commonly used in business. The problem stems from the lack of a pre-processor, any compiler directives are ignored. The second issue is the parser ignores certain parts of the code. This limitation was mainly due to time constraints on the project. The compiler problem could be solved by creating a pre-processor or by using an existing one such as M4 [GNU07].

Throughout the dissertation an emphasis on techniques and validation has been the focus. One of the big issues in call chain extraction, not evaluated in this dissertation, was that of complexity. It is quite feasible that in a real world application there exists millions of lines of code [Das02]. Though the complexity of these algorithms is discussed;

[2] The function could return a function passed to it or reference the functions in memory

an empirical study of this problem would be interesting. This leads the author to note that an interesting research area to pursue may be that of an incremental call graph extractor. Coupled with the use of dynamic graphs the possibility of analysing distinct sections of code seems quite reasonable. This could lead to a wider spread use of call graphing as the analysis could easily be performed in real-time.

Many languages have not only the same calling syntax as C but have similar semantics. Further work on the extraction framework could be carried out to allow the analysis of code from a multitude of languages. Use of the GCC compiler to generate the AST would seem appropriate as support for a number of programming languages already exists.

It is quite feasible that this project could be further developed into a useful application. Further work on the experimentation front end could lead to a useful online code analysis tool. The author sees this as being a great extension to a code repository such as a CVS or sourceforge.

Finally the problem of call graph extraction with respect to Object Oriented programming languages has not been discussed. Objected oriented languages attempt to simplify the process of creating code by abstracting code into classes [DN66]. This means that each class has a large code complexity and its own calling structure. Furthermore there exist relationships between classes in the form of polymorphism and inheritance. Thus the problem of extracting the calling structure is much more complex. Once the code has been analysed the issue of visualising the calls and structure between these classes is also extremely difficult.

10.2.2 Visualisation

The main improvement to the visualisation would be to expand the types of metadata shown on the graph. One route would be to expand the use of edges adding new calling constructs. A better method for showing when a function is called repeatedly would be useful. Another good use would be to show if the invocation emanates from a function pointer or a direct call.

Identifying the critical path through an application would also be a useful technique. The graph could visualise this route accordingly; for instance changing edges in this path to a different colour. Following from this functions could be grouped into specific sections. These groups of functions often identify a logical block of functionality and could be identified as such.

Force-Directed Graphs

One of the problems with the force-directed graphs is the physics engine. A major improvement would be to improve the physics to support more edges and nodes in a single view. This would make the dynamic view more useful; possibly replacing the static graphs. To this end a further modification would be required to support some of the more important conventions defined in the schema. This would include the ability to show broken edges and recursion.

10.3 Final Remark

This dissertation has aimed to understand the problem of call graphing and some of the difficulties in a language such as C. Various analysis techniques were looked at and implemented. The dissertation has been refined and expanded to encompass and solve problems that were initially unknown. A self-critical stance has been maintained throughout; resulting in a number of 'eureka' moments. These added to the already interesting nature of the problem. In conclusion this dissertation not only surpasses the authors original aims but also raises a number of open questions.

Bibliography

[ACT99] G. Antoniol, F. Calzolari, and P. Tonella. Impact of function pointers on the call graph. pages 51–59, 1999.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[Atk04] D.C. Atkinson. Accurate call graph extraction of programs with function pointers using type signatures. *Software Engineering Conference, 2004. 11th Asia-Pacific*, SE-5:326–335, 2004.

[ATT06] ATT. The dot language. 2006.

[ATT07] ATT. Graphviz - graph visualization software. 2007.

[BHR95] Franz-Josef Brandenburg, Michael Himsholt, and Christoph Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. In *Graph Drawing*, pages 76–87, 1995.

[Bla07] Paul E. Black. Binary search tree. *Dictionary of Algorithms and Data Structures [online]*, 2007.

[Bur01] S.; Burd, L.; Rank. Using automated source code analysis for software evolution. *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on*, pages 204–210, 2001.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. *ACM SIGPLAN Notices*, 35(5):35–46, 2000.

[Das02] Dr. Manuvir Das. Inter-procedural alias analysis. *Microsoft Internal Talk*, 2002.

[DHGN99] David P. Dobkin, Alejo Hausner, Emden R. Gansner, and Stephen C. North. Uncluttering force-directed graph layouts. In *SCG '99: Proceedings of the*

fifteenth annual symposium on Computational geometry, pages 425–426, New York, NY, USA, 1999. ACM Press.

[DN66] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Commun. ACM*, 9(9):671–678, 1966.

[EGH94a] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM Press.

[EGH94b] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, 1994.

[Fou07] Free Software Foundation. Gnu compiler collection. *http://www.gnu.org/*, 2007.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 108–124, New York, NY, USA, 1997. ACM Press.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[GNU07] GNU. Gnu m4 - unix macro processor. *http://www.gnu.org/*, 2007.

[God02] Michael W. Godfrey. Acacia and cppets. *Software Architecture Group, University of Waterloo*, 2002.

[Hil75] Murray Hill. Lex - a lexical analyzer generator. *Computer Science Technology Report*, 39, 1975.

[Hin01] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, 2001.

[HP97] M. Hind and A. Pioli. An empirical comparison of interprocedural pointer alias analyses, 1997.

[Hub04] Jan Hubicka. The gcc call graph module, 2004.

[Joh79] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[Jon05] Mr. Tim Jones. Visualize function calls with graphviz. 2005.

[KST97] Kaplan, Shamir, and Tarjan. Faster and simpler algorithm for sorting signed permutations by reversals. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1997.

[MGA92] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. Memspy: analyzing memory system bottlenecks in programs. In *SIGMETRICS '92/PERFORMANCE '92: Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 1–12, New York, NY, USA, 1992. ACM Press.

[MMG06] Richard McDougall, Jim Mauro, and Brendan Gregg. *Solaris(TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris (Solaris Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[MNGL98] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An empirical study of static call graph extractors. *ACM Trans. Softw. Eng. Methodol.*, 7(2):158–191, 1998.

[MRR04] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graphs for c programs with function pointers. *Automated Software Engg.*, 11(1):7–26, 2004.

[NAS06] NASA. Blue marble - next generation. *http://www.nasa.gov*, 2006.

[Rit93] Dennis M. Ritchie. The development of the C language. *ACM SIGPLAN Notices*, 28(3):201–208, 1993.

[RLV+96] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure

and performance of interpreters. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 150–159, New York, NY, USA, 1996. ACM Press.

[Ryd79] B.G. Ryder. Constructing the call graph of a program. *Software Engineering IEEE Transactions on*, SE-5:216–226, 1979.

[Sch06] Kyle Scholz. Jsviz - force-directed javascript graphing. 2006.

[Sha97] S. Shapiro, M. Horwitz. Fast and accurate flow-insensitive points-to analysis. *CONFERENCE RECORD OF THE ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES*, 24:1–14, 1997.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[Str06] Dr. Michelle Strout. Cs 553: Compiler construction, 2006.

[Sun03] Tao Qin; Lu Zhang; Zhiying Zhou; Dan Hao; Jiasu Sun;. Discovering use cases from source code using the branch-reserving call graph. *Software Engineering Conference, 2003. Tenth Asia-Pacific*, pages 60–67, 2003.

[ZRL96] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *Foundations of Software Engineering*, pages 81–92, 1996.

Appendix A

Project

A.0.1 Project Goals

A.1 Requirements Specification

The following is an initial list of application requirements.

- Generation of Flow Data
  - Parse the C Syntax
  - Calculate flow data from parsed C
  - Generate a human readable output of the flow
    - may correspond to existing standards
    - may extend existing standards
    - may incorporate a new system
  - Perform analysis in reasonable time
- Creation of Graphs
  - Parse the flow data
  - Create state graphs based on the flow data
  - Detect the important paths through code
  - Detect superfluous flow data

- Detect function cycles
- Dynamically generate graphs
- Allow various levels of graph viewing
- Allow user to query for specific sections of code

A.2 Software Life Cycle

The following application will be developed using an evolutionary lifecycle. As most of the requirements can have initial implementations which lack certain functionality but can be built on, this is the logical methodology to take.

A.3 Project Plan

A.3.1 Milestones

- Stage 1: Flow Data
  - Parse C Grammar
  - Define format for flow data
  - Outputting flow data
- Stage 2: Call Graph
  - Parse flow data
  - Create simple Call Graphs
  - Calculate superfluous calls
  - Calculate graph cycles
  - Allow dynamic viewing of graphs
  - Link graphs to source code
  - Compare and contrast existing systems
- Stage 3: Project Conclusion

A.3.2 Endpoints

- Stage 1: Flow Data
  - Parser
    - Simple C Parser (End Point)
    - Complete C Parser (End Point)

  - Flow Data
    - Implement Existing syntax (End Point)
    - Extend/re-implement syntax (End Point)

- Stage 2: Graph generator
  - Graphs
    - Simple Graphs (End Point)
    - Complex Graphs (End Point)
    - Dynamic Graphs (End Point)
  - Parsing
    - Simple Parser (End Point)
    - Calculate important flows (End Point)
    - Calculate function cycles (End Point)

A.3.3 Project Deliverables

- Project Proposal: 23/10/2006
- Literary review: 11/12/2006
- Progress check form: 05/02/2007
- Poster presentation: 19/03/2007
- Dissertation (Final report): 30/04/2007

A.3.4 Project Goals

- Project Proposal: 20/10/2006
- Literature survey: 30/10/2006
- Parser and Output method: 15/11/2006
- Write-up of Parsing method: 20/11/2006
- Simplistic Graph Generation: 08/12/2006
- Dynamic Graph Generation: 01/03/2007

- Create Poster Presentation: 09/03/2007
- Graph Algorithm documentation 15/03/2007
- Complete documentation: 30/03/2007

A.4 Resources

A.4.1 Equipment

- Computer
- Printer

As this application is mainly software based, little equipment is needed: A computer is necessary to create the application as well as test it. A printer is a useful resource for printing the call graphs as well as papers associated with the application.

A.4.2 Software

- Development environment
- Compiler
- Word Processor & Latex
- GCC (UNIX)
- Addr2Line
- Existing Call Graph Programs such as CodeViz and GraphViz

A number of applications are needed to compete this project. From the implementation side a tried and tested development environment 1 and Compiler 2 are needed to create this application. Preferences are for a cross platform language making the tool useful for *nix and Windows users, Java and with the advent of MONO C# are likely choices. GCC is needed to test that applications compile and is a necessary component of some call graph generators. Addd2Line is an application that can take memory calls inside an application and convert them to lines of code, highly useful for call graph generation. A number of existing call graph generators are available these will be used for comparison purposes.

A.5 Risk Assessment

- Requirements Change
  - High Probability: Due to the evolutionary design model I have taken, at this stage requirements are highly likely to change. As this is seen as part of the development process, core goals will remain static.
- Illness
  - Medium Probability: There is a good change at some time during this year I will be ill, though with appropriate planning and working ahead of schedule this can be handled.
- Loss of work
  - Low Probability: Use of content versioning system as well as backups of files and source code to different geological locations should prove sufficient.
- Unrealistic requirements
  - Low: During the design phase realistic requirements will be chosen and checked with the project supervisor

A.6 Empirical Results of Experiments

The following is a listing of the experiments carried out in Section 8 (page 73). The results of the algorithm are rated firstly on if the output was as expected. Secondly the correctness, how closely the extraction reflects the code, is chosen. The word *yes* implies the result was as expected or the results were correct. Otherwise the output was incorrect or unexpected.

	Expected Output	Correct
Experiment 1 - Naive	Yes	Yes
Steensgard	Yes	Yes
Emami	Yes	Yes
Experiment 2 - Naive	Yes	Yes
Steensgard	Yes	Yes
Emami	Yes	Yes
Experiment 3 - Naive	Yes	Yes
Steensgard	Yes	Yes
Emami	Yes	Yes
Experiment 4 - Naive	Yes	Yes
Steensgard	Yes	Yes
Emami	Yes	Yes
Experiment 5 - Naive	Yes	**No**
Steensgard	Yes	Yes
Emami	Yes	Yes
Experiment 6 - Naive	Yes	**No**
Steensgard	Yes	Yes
Emami	Yes	Yes
Experiment 7 - Naive	Yes	**No**
Steensgard	Yes	**No**
Emami	Yes	Yes
Experiment 8 - Naive	Yes	**No**
Steensgard	Yes	Yes
Emami	Yes	Yes

	Expected Output	Correct
Experiment 9 - Naive	Yes	**No**
Steensgard	Yes	**No**
Emami	Yes	Yes
Experiment 10 - Naive	Yes	**No**
Steensgard	Yes	Yes
Emami	Yes	Yes
Experiment 11 - Naive	Yes	**No**
Steensgard	Yes	**No**
Emami	Yes	Yes
Experiment 12 - Naive	Yes	**No**
Steensgard	Yes	**No**
Emami	Yes	Yes
Experiment 13 - Naive	Yes	**No**
Steensgard	Yes	**No**
Emami	Yes	Yes
Experiment 14 - Naive	Yes	**No**
Steensgard	Yes	Yes
Emami	Yes	Yes
Experiment 15 - Naive	Yes	**No**
Steensgard	Yes	**No**
Emami	Yes	Yes
Experiment 16 - Naive	Yes	Yes
Steensgard	Yes	Yes
Emami	Yes	Yes

A.7 Algorithms and their Analysis Attributes

Algorithm	Flow Senstive	Context Senstive	Field Sensitive	Inter-procedural
Naive Algorithm	False	False	False	False
Steensgard's Algorithm	False	False	False	False
Emami's Algorithm	True	True	False	True

Figure A.1: Call Graph Analysis Methods and Attributes of each

Appendix B

Source Code - Extraction

B.1 Source code for C.flex

```
%{
    /* Copyright (C) 1989,1990 James A. Roskind, All rights reserved.
    This grammar was developed and written by James A. Roskind.
    Copying of this grammar description, as a whole, is permitted
    providing this notice is intact and applicable in all complete
    copies. Translations as a whole to other parser generator input
    languages (or grammar description languages) is permitted
    provided that this notice is intact and applicable in all such
    copies, along with a disclaimer that the contents are a
    translation. The reproduction of derived text, such as modified
    versions of this grammar, or the output of parser generators, is
    permitted, provided the resulting work includes the copyright
    notice "Portions Copyright (c) 1989, 1990 James A. Roskind".
```

```
14      Derived products, such as compilers, translators, browsers, etc.,
15      that use this grammar, must also provide the notice "Portions
16      Copyright (c) 1989, 1990 James A. Roskind" in a manner
17      appropriate to the utility, and in keeping with copyright law
18      (e.g.: EITHER displayed when first invoked/executed; OR displayed
19      continuously on display terminal; OR via placement in the object
20      code in form readable in a printout, with or near the title of
21      the work, or at the end of the file). No royalties, licenses or
22      commissions of any kind are required to copy this grammar, its
23      translations, or derivative products, when the copies are made in
24      compliance with this notice. Persons or corporations that do make
25      copies in compliance with this notice may charge whatever price
26      is agreeable to a buyer, for such copies or derivative works.
27      THIS GRAMMAR IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR
28      IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
29      WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
30      PURPOSE.
31
32      James A. Roskind
33      Independent Consultant
34      516 Latania Palm Drive
35      Indialantic FL, 32903
36      (407)729-4348
37      jar@ileaf.com
38      or ...!uunet!leafusa!jar
39
40      ----end of copyright notice----
41      */
42
43  /* Included code before lex code */
44  /************** Includes and Defines ****************************/
45  #include "C.tab.h" /* YACC generated definitions based on C++ parser input*/
```

```
46
47 #include <stdio.h>
48
49 extern TOKEN* lookup_token(char*);
50
51 TOKEN* make_string(char*);
52 TOKEN* make_int(char*);
53 TOKEN* lasttok;
54
55 void count(void);
56 void comment(void);
57 void comp_directive(void);
58
59 /* modified */
60 /* typedef char * YYSTYPE; /* interface with lexer: should be in header file*/
61
62 /* char * yylval; /* We will always point at the text of the lexeme.
63     This makes it easy to print out nice trees when YYDEBUG is
64     enabled (see the C++ grammar file, and its definition of
65     YYDEBUG_LEXER_TEXT to be "yylval") */
66
67
68 #define WHITE_RETURN(x)  /* do nothing, */
69
70 #define NEW_LINE_RETURN() WHITE_RETURN('\n')
71
72 #define PA_KEYWORD_RETURN(x)    RETURN_VAL(x)  /* standard C PArser Keyword */
73 #define CPP_KEYWORD_RETURN(x)   PA_KEYWORD_RETURN(x)  /* C++ keyword */
74 #define PPPA_KEYWORD_RETURN(x) RETURN_VAL(x)  /* both PreProcessor and PArser keyword */
75 #define PP_KEYWORD_RETURN(x)    IDENTIFIER_RETURN()
76
77 #define IDENTIFIER_RETURN() RETURN_VAL(isaTYPE(yytext)?TYPEDEFname:IDENTIFIER)
```

```
78
79 #define PPOP_RETURN(x)        RETURN_VAL((int)*yytext) /* PreProcess and Parser operator */
80 #define NAMED_PPOP_RETURN(x) /* error: PreProcessor ONLY operator;  Do nothing */
81 #define ASCIIOP_RETURN(x)     RETURN_VAL((int)*yytext) /* a single character operator */
82 #define NAMEDOP_RETURN(x)     RETURN_VAL(x)             /* a multichar operator, with a name */
83
84 #define NUMERICAL_RETURN(x) RETURN_VAL(x)             /* some sort of constant */
85 #define LITERAL_RETURN(x)   RETURN_VAL(x)             /* a string literal */
86
87 #define RETURN_VAL(x) yylval = yytext; return(x);
88
89
90 %}
91
92 /*identifier [a-zA-Z_][0-9a-zA-Z_]**/
93 identifier [A-Za-z_]+[0-9A-Za-z_]*
94
95
96 exponent_part [eE][-+]?[0-9]+
97 fractional_constant ([0-9]*"."[0-9]+)|([0-9]+".")
98 floating_constant (({fractional_constant}{exponent_part}?)|([0-9]+{exponent_part}))[FfLl]?
99
100 integer_suffix_opt ([uU]?[lL]?)|([lL][uU])
101 decimal_constant [1-9][0-9]*{integer_suffix_opt}
102 octal_constant "0"[0-7]*{integer_suffix_opt}
103 hex_constant "0"[xX][0-9a-fA-F]+{integer_suffix_opt}
104
105 simple_escape [abfnrtv'"?\\]
106 octal_escape  [0-7]{1,3}
107 hex_escape "x"[0-9a-fA-F]+
108
109 escape_sequence [\\]({simple_escape}|{octal_escape}|{hex_escape})
```

```
110 c_char [^'\\\n]|{escape_sequence}
111 s_char [^"\\\n]|{escape_sequence}
112
113 h_tab [\011]
114 form_feed [\014]
115 v_tab [\013]
116 c_return [\015]
117
118 horizontal_white [ ]|{h_tab}
119
120 %%
121
122 "#"                 { comp_directive(); }
123 "/*"            { comment(); }
124
125 {horizontal_white}+     {
126                         WHITE_RETURN(' ');
127                         }
128
129 ({v_tab}|{c_return}|{form_feed})+   {
130                         WHITE_RETURN(' ');
131                         }
132
133
134 ({horizontal_white}|{v_tab}|{c_return}|{form_feed})*"\n"   {
135                         NEW_LINE_RETURN();
136                         }
137
138 "auto"              {count(); return(AUTO);}
139 "break"             {count(); return(BREAK);}
140 "case"              {count(); return(CASE);}
141 "char"              {count(); return(CHAR);}
```

```
142  "const"          {count(); return(CONST);}
143  "continue"       {count(); return(CONTINUE);}
144  "default"        {count(); return(DEFAULT);}
145  define           {count(); PP_KEYWORD_RETURN(DEFINE);}      /* compiler directive */
146  defined          {count(); PP_KEYWORD_RETURN(OPDEFINED);}     /* compiler directive */
147  "do"             {count(); return(DO);}
148  "double"         {count(); return(DOUBLE);}
149  elif             {count(); PP_KEYWORD_RETURN(ELIF);}      /* compiler directive */
150  "else"           {count(); return(ELSE);}
151  endif            {count(); PP_KEYWORD_RETURN(ENDIF);}      /* compiler directive */
152  "enum"           {count(); return(ENUM);}
153  error            {count(); PP_KEYWORD_RETURN(ERROR);}      /* compiler directive */
154  "extern"         {count(); return(EXTERN);}
155  "float"          {count(); return(FLOAT);}
156  "for"            {count(); return(FOR);}
157  "goto"           {count(); return(GOTO);}
158  "if"             {count(); return(IF);}
159  ifdef            {count(); PP_KEYWORD_RETURN(IFDEF);}      /* compiler directive */
160  ifndef           {count(); PP_KEYWORD_RETURN(IFNDEF);}      /* compiler directive */
161  include          {count(); PP_KEYWORD_RETURN(INCLUDE); }     /* compiler directive */
162  "int"            {count(); return(INT);}
163  line             {count(); PP_KEYWORD_RETURN(LINE);}      /* compiler directive */
164  "long"           {count(); return(LONG);}
165  pragma           {count(); PP_KEYWORD_RETURN(PRAGMA);}      /* compiler directive */
166  "register"       {count(); return(REGISTER);}
167  "return"         {count(); return(RETURN);}
168  "short"          {count(); return(SHORT);}
169  "signed"         {count(); return(SIGNED);}
170  "sizeof"         {count(); return(SIZEOF);}
171  "static"         {count(); return(STATIC);}
172  "struct"         {count(); return(STRUCT);}
173  "switch"         {count(); return(SWITCH);}
```

```
174 "typedef"            {count(); return(TYPEDEF);}
175 undef                {count(); PP_KEYWORD_RETURN(UNDEF);}      /* compiler directive */
176 "union"              {count(); return(UNION);}
177 "unsigned"           {count(); return(UNSIGNED);}
178 "void"               {count(); return(VOID);}
179 "volatile"           {count(); return(VOLATILE);}
180 "while"              {count(); return(WHILE);}
181
182 {identifier}         {count(); lasttok = lookup_token(yytext); IDENTIFIER_RETURN();}
183
184 {decimal_constant}   {count(); lasttok = make_int(yytext); NUMERICAL_RETURN(INTEGERconstant);}
185 {octal_constant}     {count(); NUMERICAL_RETURN(OCTALconstant);}
186 {hex_constant}       {count(); NUMERICAL_RETURN(HEXconstant);}
187 {floating_constant}  {count(); NUMERICAL_RETURN(FLOATINGconstant);}
188
189
190 "L"?['][c_char]+[']      {count();
191                           NUMERICAL_RETURN(CHARACTERconstant);
192                           }
193
194
195 "L"?["][s_char]*["]      {count(); lasttok = make_string(yytext);
196                           LITERAL_RETURN(STRINGliteral);}
197
198
199 "("                  {count(); return('(');}
200 ")"                  {count(); return(')');}
201 ","                  {count(); return(',');}
202 "#"                  {count(); return('#');}
203 "##"                 {count(); return('#');} /* problem */
204
205 "{"                  {count(); return('{');}
```

```
206 "}"                 {count(); return('}');}
207 "["                 {count(); return('[');}
208 "]"                 {count(); return(']');}
209 "."                 {count(); return('.');}
210 "&"                 {count(); return('&');}
211 "*"                 {count(); return('*');}
212 "+"                 {count(); return('+');}
213 "-"                 {count(); return('-');}
214 "~"                 {count(); return('~');}
215 "!"                 {count(); return('!');}
216 "/"                 {count(); return('/');}
217 "%"                 {count(); return('%');}
218 "<"                 {count(); return('<');}
219 ">"                 {count(); return('>');}
220 "^"                 {count(); return('^');}
221 "|"                 {count(); return('|');}
222 "?"                 {count(); return('?');}
223 ":"                 {count(); return(':');}
224 ";"                 {count(); return(';');}
225 "="                 {count(); return('=');}
226
227
228 "->"                {count(); NAMEDOP_RETURN(ARROW);}
229 "++"                {count(); NAMEDOP_RETURN(ICR);}
230 "--"                {count(); NAMEDOP_RETURN(DECR);}
231 "<<"                {count(); NAMEDOP_RETURN(LS);}
232 ">>"                {count(); NAMEDOP_RETURN(RS);}
233 "<="                {count(); NAMEDOP_RETURN(LE);}
234 ">="                {count(); NAMEDOP_RETURN(GE);}
235 "=="                {count(); NAMEDOP_RETURN(EQ);}
236 "!="                {count(); NAMEDOP_RETURN(NE);}
237 "&&"                {count(); NAMEDOP_RETURN(ANDAND);}
```

```
238 "||"                    {count(); NAMEDOP_RETURN(OROR);}
239 "*="                    {count(); NAMEDOP_RETURN(MULTassign);}
240 "/="                    {count(); NAMEDOP_RETURN(DIVassign);}
241 "%="                    {count(); NAMEDOP_RETURN(MODassign);}
242 "+="                    {count(); NAMEDOP_RETURN(PLUSassign);}
243 "-="                    {count(); NAMEDOP_RETURN(MINUSassign);}
244 "<<="                   {count(); NAMEDOP_RETURN(LSassign);}
245 ">>="                   {count(); NAMEDOP_RETURN(RSassign);}
246 "&="                    {count(); NAMEDOP_RETURN(ANDassign);}
247 "^="                    {count(); NAMEDOP_RETURN(ERassign);}
248 "|="                    {count(); NAMEDOP_RETURN(ORassign);}
249 "..."                   {count(); NAMEDOP_RETURN(ELLIPSIS);}
250
251
252 [\t\v\n\f]          { count(); }
253 .             { /* ignore bad characters */ }
254
255 %%
```

B.2 Source code for C.y

```
%{
/* Copyright (C) 1989,1990 James A. Roskind, All rights reserved.
This grammar was developed and written by James A. Roskind.
Copying of this grammar description, as a whole, is permitted
providing this notice is intact and applicable in all complete
copies. Translations as a whole to other parser generator input
languages (or grammar description languages) is permitted
provided that this notice is intact and applicable in all such
copies, along with a disclaimer that the contents are a
translation. The reproduction of derived text, such as modified
versions of this grammar, or the output of parser generators, is
permitted, provided the resulting work includes the copyright
notice "Portions Copyright (c) 1989, 1990 James A. Roskind".
Derived products, such as compilers, translators, browsers, etc.,
that use this grammar, must also provide the notice "Portions
Copyright (c) 1989, 1990 James A. Roskind" in a manner
appropriate to the utility, and in keeping with copyright law
(e.g.: EITHER displayed when first invoked/executed; OR displayed
continuously on display terminal; OR via placement in the object
code in form readable in a printout, with or near the title of
the work, or at the end of the file). No royalties, licenses or
commissions of any kind are required to copy this grammar, its
translations, or derivative products, when the copies are made in
compliance with this notice. Persons or corporations that do make
copies in compliance with this notice may charge whatever price
is agreeable to a buyer, for such copies or derivative works.
THIS GRAMMAR IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.
```

```
31
32      James A. Roskind
33      Independent Consultant
34      516 Latania Palm Drive
35      Indialantic FL, 32903
36      (407)729-4348
37      jar@ileaf.com
38      or ...!uunet!leafusa!jar
39
40      ----end of copyright notice----
41      */
42
43  #define YYSTYPE NODE*
44  #define YYDEBUG 1
45
46
47  extern TOKEN *int_token, *void_token, *lasttok;
48
49
50  NODE *ans;
51  %}
52
53  /* Define terminal tokens */
54  /* keywords */
55  %token AUTO             DOUBLE          INT             STRUCT
56  %token BREAK            ELSE            LONG            SWITCH
57  %token CASE             ENUM            REGISTER        TYPEDEF
58  %token CHAR             EXTERN          RETURN          UNION
59  %token CONST            FLOAT           SHORT           UNSIGNED
60  %token CONTINUE         FOR             SIGNED          VOID
61  %token DEFAULT          GOTO            SIZEOF          VOLATILE
62  %token DO               IF              STATIC          WHILE
```

```
63
64 %token APPLY LEAF
65
66 /* ANSI Grammar suggestions */
67 %token IDENTIFIER                STRINGliteral
68 %token FLOATINGconstant          INTEGERconstant          CHARACTERconstant
69 %token OCTALconstant             HEXconstant
70
71 /* New Lexical element, whereas ANSI suggested non-terminal */
72
73 %token TYPEDEFname /* Lexer will tell the difference between this and
74     an identifier!  An identifier that is CURRENTLY in scope as a
75     typedef name is provided to the parser as a TYPEDEFname.*/
76
77 /* Multi-Character operators */
78 %token  ARROW            /*    ->                                    */
79 %token  ICR DECR         /*    ++      --                            */
80 %token  LS RS            /*    <<      >>                            */
81 %token  LE GE EQ NE      /*    <=      >=      ==      !=            */
82 %token  ANDAND OROR      /*    &&      ||                            */
83 %token  ELLIPSIS         /*    ...                                   */
84
85 /* modifying assignment operators */
86 %token MULTassign  DIVassign   MODassign   /*  *=    /=    %=    */
87 %token PLUSassign  MINUSassign             /*  +=    -=          */
88 %token LSassign    RSassign                /*  <<=   >>=         */
89 %token ANDassign   ERassign    ORassign    /*  &=    ^=    |=    */
90
91 %start prog.start
92
93 %%
94 prog.start
```

```
95          : translation.unit  { ans = $$ = $1;}
96          ;
97
98  /* CONSTANTS */
99  constant
100         :   FLOATINGconstant     { $$ = make_leaf(lasttok); }
101         |   INTEGERconstant      { $$ = make_leaf(lasttok); }
102         |   OCTALconstant        { $$ = make_leaf(lasttok); }
103         |   HEXconstant          { $$ = make_leaf(lasttok); }
104         |   CHARACTERconstant    { $$ = make_leaf(lasttok); }
105         ;
106
107 /* STRING LITERALS */
108 string.literal.list
109         : STRINGliteral { $$ = make_leaf(lasttok); }
110         | string.literal.list ',' STRINGliteral { $$ = make_node(',', $1, $3); }
111         ;
112
113
114 /* EXPRESSIONS */
115 primary.expression
116         : IDENTIFIER          { $$ = make_leaf(lasttok); } /* We cannot use a typedef name as a variable */
117         | constant            { $$ = $1; }
118         | string.literal.list { $$ = $1; }
119         | '(' expression ')'  { $$ = $2; }
120         ;
121
122 postfix.expression
123         : primary.expression       { $$ = $1; }
124         | postfix.expression '[' expression ']'  { }
125         | postfix.expression '(' ')'    { $$ = make_node(APPLY, $1, NULL); }
126         | postfix.expression '(' argument.expression.list ')'  {  $$ = make_node(APPLY, $1, $3); }
```

```
127         | postfix.expression '.' identifier.or.typedef.name    {    }
128         | postfix.expression ARROW identifier.or.typedef.name  {    }
129         | postfix.expression ICR
130         | postfix.expression DECR
131         ;
132
133 argument.expression.list
134         : assignment.expression { $$ = $1; }
135         | argument.expression.list ',' assignment.expression  {  $$ = make_node(',', $1, $3); }
136         ;
137
138 unary.expression
139         : postfix.expression { $$ = $1; }
140         | ICR unary.expression
141         | DECR unary.expression
142         | unary.operator cast.expression  { $$ = make_node((int)$1, $2, NULL); }
143         | SIZEOF unary.expression
144         | SIZEOF '(' type.name ')'
145         ;
146
147 unary.operator
148         : '&'         { $$ = (void*)'&'; }
149         | '*'         { $$ = (void*)'*'; }
150         | '+'         { $$ = (void*)'+'; }
151         | '-'         { $$ = (void*)'-'; }
152         | '~'         { $$ = (void*)'~'; }
153         | '!'         { $$ = (void*)'!'; }
154         ;
155
156 cast.expression
157         :  unary.expression          { $$ = $1; }
158         | '(' type.name ')' cast.expression { $$ = $4; }
```

```
159         ;
160
161 multiplicative.expression
162         : cast.expression  { $$ = $1 }
163         | multiplicative.expression '*' cast.expression {
164                                         $$ = make_node('*', $1, $3); }
165         | multiplicative.expression '/' cast.expression {
166                                         $$ = make_node('/', $1, $3); }
167         | multiplicative.expression '%' cast.expression {
168                                         $$ = make_node('%', $1, $3); }
169         ;
170
171 additive.expression
172         : multiplicative.expression { $$ = $1; }
173         | additive.expression '+' multiplicative.expression {
174                                         $$ = make_node('+', $1, $3); }
175         | additive.expression '-' multiplicative.expression {
176                                         $$ = make_node('-', $1, $3); }
177         ;
178
179 shift.expression
180         : additive.expression         { $$ = $1; }
181         | shift.expression LS additive.expression
182         | shift.expression RS additive.expression
183         ;
184
185 relational.expression
186         : shift.expression        {  $$ = $1; }
187         | relational.expression '<' shift.expression {
188                                         $$ = make_node('<', $1, $3); }
189         | relational.expression '>' shift.expression {
190                                         $$ = make_node('>', $1, $3); }
```

```
191         | relational.expression LE shift.expression {
192                                         $$ = make_node(LE, $1, $3); }
193         | relational.expression GE shift.expression {
194                                         $$ = make_node(GE, $1, $3); }
195         ;
196
197 equality.expression
198         : relational.expression       { $$ = $1; }
199         | equality.expression EQ relational.expression {
200                                         $$ = make_node(EQ, $1, $3); }
201         | equality.expression NE relational.expression {
202                                         $$ = make_node(NE, $1, $3); }
203         ;
204
205 AND.expression
206         : equality.expression       { $$ = $1; }
207         | AND.expression '&' equality.expression  {
208                                         $$ = make_node('&', $1, $3); }
209         ;
210
211 exclusive.OR.expression
212         : AND.expression       { $$ = $1; }
213         | exclusive.OR.expression '^' AND.expression {
214                                         $$ = make_node('^', $1, $3); }
215         ;
216
217 inclusive.OR.expression
218         : exclusive.OR.expression       { $$ = $1; }
219         | inclusive.OR.expression '|' exclusive.OR.expression {
220                                         $$ = make_node('|', $1, $3); }
221         ;
222
```

```
223 logical.AND.expression
224         : inclusive.OR.expression        { $$ = $1; }
225         | logical.AND.expression ANDAND inclusive.OR.expression {
226                                             $$ = make_node(ANDAND, $1, $3); }
227         ;
228
229 logical.OR.expression
230         : logical.AND.expression         {   $$ = $1; }
231         | logical.OR.expression OROR logical.AND.expression {
232                                             $$ = make_node(OROR, $1, $3); }
233         ;
234
235 conditional.expression
236         : logical.OR.expression         {   $$ = $1; }
237         | logical.OR.expression '?' expression ':'
238             conditional.expression
239         ;
240
241 assignment.expression
242         : conditional.expression         { $$ = $1; }
243         | unary.expression assignment.operator assignment.expression  {
244                                             $$ = make_node('=', $1, $3); }
245         ;
246
247 assignment.operator:
248         '='
249         | MULTassign          { $$ = $1; }
250         | DIVassign           { $$ = $1; }
251         | MODassign           { $$ = $1; }
252         | PLUSassign          { $$ = $1; }
253         | MINUSassign         { $$ = $1; }
254         | LSassign            { $$ = $1; }
```

```
255         | RSassign              { $$ = $1; }
256         | ANDassign             { $$ = $1; }
257         | ERassign              { $$ = $1; }
258         | ORassign              { $$ = $1; }
259         ;
260
261 expression
262         : assignment.expression     { $$ = $1; }
263         | expression ',' assignment.expression { $$ = make_node(',', $1, $3); }
264         ;
265
266 constant.expression
267         : conditional.expression     {  $$ = $1; }
268         ;
269
270     /* The following was used for clarity */
271 expression.opt:
272         /* Nothing */ { $$ = NULL; }
273         | expression { $$ = $1; }
274         ;
275
276
277
278 /* DECLARATIONS */
279
280     /* The following is different from the ANSI C specified  grammar.
281     The changes were made to disambiguate typedef's presence in
282     declaration.specifiers (vs. in the declarator for redefinition);
283     to allow struct/union/enum tag declarations without declarators,
284     and to better reflect the parsing of declarations (declarators
285     must be combined with declaration.specifiers ASAP so that they
286     are visible in scope).
```

```
287
288     Example of typedef use as either a declaration.specifier or a
289     declarator:
290
291       typedef int T;
292       struct S { T T;}; /* redefinition of T as member name */
293
294     Example of legal and illegal statements detected by this grammar:
295
296       int; /* syntax error: vacuous declaration */
297       struct S;  /* no error: tag is defined or elaborated */
298
299     Example of result of proper declaration binding:
300
301         int a=sizeof(a); /* note that "a" is declared with a type  in
302             the name space BEFORE parsing the initializer */
303
304         int b, c[sizeof(b)]; /* Note that the first declarator "b" is
305              declared with a type BEFORE the second declarator is
306              parsed */
307
308     */
309
310 declaration
311         : sue.declaration.specifier ';'         { $$ = $1; }
312         | sue.type.specifier ';'                { $$ = $1; }
313         | declaring.list ';'                    { $$ = $1; }
314         | default.declaring.list ';'            { $$ = $1; }
315         ;
316
317     /* Note that if a typedef were redeclared, then a declaration
318     specifier must be supplied */
```

```
319
320 default.declaring.list  /* Can't  redeclare typedef names */
321         : declaration.qualifier.list identifier.declarator {} initializer.opt  {}
322         | type.qualifier.list identifier.declarator {} initializer.opt          {}
323         | default.declaring.list ',' identifier.declarator {} initializer.opt  {}
324         ;
325
326 declaring.list
327         : declaration.specifier declarator {} initializer.opt {  }
328         | type.specifier declarator {} initializer.opt {
329                                                         if($4 == NULL)
330                                                         {
331                                                            $$ = make_node('~', $1, $2);
332                                                         }
333                                                           else
334                                                         {
335                                                            $$ = make_node('~', $1,make_node('=',$2,$4));
336                                                         }
337                                                       }
338
339         | declaring.list ',' declarator {} initializer.opt { }
340         ;
341
342 declaration.specifier
343         : basic.declaration.specifier        { $$ = $1; } /* Arithmetic or void */
344         | sue.declaration.specifier          { $$ = $1; } /* struct/union/enum */
345         | typedef.declaration.specifier      { $$ = $1; } /* typedef*/
346         ;
347
348 type.specifier
349         : basic.type.specifier               { $$ = $1; } /* Arithmetic or void */
350         | sue.type.specifier                 { $$ = $1; } /* Struct/Union/Enum */
```

```
351         | typedef.type.specifier              { $$ = $1; } /* Typedef */
352         ;
353
354
355 declaration.qualifier.list  /* const/volatile, AND storage class */
356         : storage.class      { $$ = $1; }
357         | type.qualifier.list storage.class { $$ = $1; }
358         | declaration.qualifier.list declaration.qualifier {  }
359         ;
360
361 type.qualifier.list
362         : type.qualifier   { $$ = $1; }
363         | type.qualifier.list type.qualifier  { /* printf("type.qualifier.list  - 2"); */ }
364         ;
365
366 declaration.qualifier
367         : type.qualifier             { $$ = $1; }  /* const or volatile */
368         | storage.class              { $$ = $1; }
369         ;
370
371 type.qualifier
372         : CONST             { $$ = $1; }
373         | VOLATILE          { $$ = $1; }
374         ;
375
376 basic.declaration.specifier  /*StorageClass+Arithmetic or void*/
377         : basic.type.specifier  storage.class
378         | declaration.qualifier.list basic.type.name
379         | basic.declaration.specifier declaration.qualifier
380         | basic.declaration.specifier basic.type.name
381         ;
382
```

```
383 basic.type.specifier /* Arithmetic or void */
384         : basic.type.name     {  $$ = $1 }
385         | type.qualifier.list basic.type.name
386         | basic.type.specifier type.qualifier
387         | basic.type.specifier basic.type.name
388         ;
389
390 sue.declaration.specifier           /* StorageClass + struct/union/enum */
391         : sue.type.specifier storage.class
392         | declaration.qualifier.list elaborated.type.name
393         | sue.declaration.specifier declaration.qualifier
394         ;
395
396 sue.type.specifier    /* struct/union/enum */
397         : elaborated.type.name
398         | type.qualifier.list elaborated.type.name
399         | sue.type.specifier type.qualifier
400         ;
401
402
403 typedef.declaration.specifier       /*Storage Class + typedef types */
404         : typedef.type.specifier storage.class
405         | declaration.qualifier.list TYPEDEFname
406         | typedef.declaration.specifier declaration.qualifier
407         ;
408
409 typedef.type.specifier              /* typedef types */
410         : TYPEDEFname     { $$ = make_leaf(void_token);  }
411         | type.qualifier.list TYPEDEFname { }
412         | typedef.type.specifier type.qualifier {  }
413         ;
414
```

```
storage.class
        : TYPEDEF
        | EXTERN
        | STATIC
        | AUTO
        | REGISTER
        ;

basic.type.name
        : VOID  { $$ = make_leaf(void_token); }
        | CHAR   { $$ = make_leaf(void_token); }
        | SHORT  { $$ = make_leaf(void_token); }
        | INT  {  $$ = make_leaf(int_token); }
        | LONG  { $$ = make_leaf(void_token); }
        | FLOAT  { $$ = make_leaf(void_token); }
        | DOUBLE  { $$ = make_leaf(void_token); }
        | SIGNED  { $$ = make_leaf(void_token); }
        | UNSIGNED  { $$ = make_leaf(void_token); }
        ;

elaborated.type.name
        : struct.or.union.specifier
        | enum.specifier
        ;

struct.or.union.specifier:
        struct.or.union '{' struct.declaration.list '}'
        | struct.or.union identifier.or.typedef.name
                '{' struct.declaration.list '}'
        | struct.or.union identifier.or.typedef.name
        ;

```

```
447 struct.or.union
448         : STRUCT
449         | UNION
450         ;
451
452 struct.declaration.list
453         : struct.declaration
454         | struct.declaration.list struct.declaration
455         ;
456
457 struct.declaration
458         : struct.declaring.list ';'
459         | struct.default.declaring.list ';'
460         ;
461
462 struct.default.declaring.list         /* doesn't redeclare typedef*/
463         : type.qualifier.list struct.identifier.declarator
464         | struct.default.declaring.list ',' struct.identifier.declarator
465         ;
466
467 struct.declaring.list
468         : type.specifier struct.declarator
469         | struct.declaring.list ',' struct.declarator
470         ;
471
472
473 struct.declarator
474         : declarator bit.field.size.opt
475         | bit.field.size
476         ;
477
478 struct.identifier.declarator
```

```
479         : identifier.declarator bit.field.size.opt
480         | bit.field.size
481         ;
482
483 bit.field.size.opt:
484         /* nothing */
485         | bit.field.size
486         ;
487
488 bit.field.size
489         : ':' constant.expression
490         ;
491
492 enum.specifier
493         : ENUM '{' enumerator.list '}'
494         | ENUM identifier.or.typedef.name '{' enumerator.list '}'
495         | ENUM identifier.or.typedef.name
496         ;
497
498
499
500 enumerator.list
501         : identifier.or.typedef.name enumerator.value.opt
502         | enumerator.list ',' identifier.or.typedef.name enumerator.value.opt
503         ;
504
505 enumerator.value.opt:
506         /* Nothing */
507         | '=' constant.expression
508         ;
509
510 parameter.type.list
```

```
511         : parameter.list { $$ = $1; }
512         | parameter.list ',' ELLIPSIS { $$ = make_node(',', $1, $3); }
513         ;
514
515 parameter.list
516         : parameter.declaration { $$ = $1; }
517         | parameter.list ',' parameter.declaration { $$ = make_node('~', $1, $3); }
518         ;
519
520 parameter.declaration
521         : declaration.specifier     {  $$ = $1; }
522         | declaration.specifier abstract.declarator { $$ = make_node('~', $1, $2); }
523         | declaration.specifier identifier.declarator { $$ = make_node('~', $1, $2); }
524         | declaration.specifier parameter.typedef.declarator { }
525         | declaration.qualifier.list    {    $$ = $1; }
526         | declaration.qualifier.list abstract.declarator { }
527         | declaration.qualifier.list identifier.declarator { }
528         | type.specifier  { $$ = $1; }
529         | type.specifier abstract.declarator { $$ = make_node('$', $1, $2); }
530         | type.specifier identifier.declarator { $$ = make_node('~', $1, $2); }
531         | type.specifier parameter.typedef.declarator {  $$ = make_node('~', $1, $2); }
532         | type.qualifier.list  { $$ = $1; }
533         | type.qualifier.list abstract.declarator { $$ = make_node('~', $1, $2); }
534         | type.qualifier.list identifier.declarator {  $$ = make_node('~', $1, $2); }
535         ;
536
537     /* ANSI C section 3.7.1 states "An identifier declared as a
538     typedef name shall not be redeclared as a parameter". Hence the
539     following is based only on IDENTIFIERs */
540
541 identifier.list
542         : IDENTIFIER      { $$ = make_leaf(lasttok); }
```

```
543         | identifier.list ',' IDENTIFIER   { $$ = make_node(',', $1, $3); }
544         ;
545
546 identifier.or.typedef.name
547         : IDENTIFIER  {  $$ = make_leaf(lasttok); }
548         | TYPEDEFname {  $$ = make_leaf(lasttok); }
549         ;
550
551 type.name
552         : type.specifier   { $$ = $1; }
553         | type.specifier abstract.declarator
554         | type.qualifier.list   { $$ = $1; }
555         | type.qualifier.list abstract.declarator
556         ;
557
558 initializer.opt:
559         /* nothing */ { $$ = NULL; }
560         | '=' initializer   { $$ = $2;}
561         ;
562
563 initializer
564         : '{' initializer.list '}'
565         | '{' initializer.list ',' '}'
566         | assignment.expression
567         ;
568
569 initializer.list
570         : initializer
571         | initializer.list ',' initializer
572         ;
573
574
```

```
575 /* STATEMENTS */
576 statement
577         : labeled.statement             { $$ = $1; }
578         | compound.statement          { $$ = $1; }
579         | expression.statement          { $$ = $1; }
580         | selection.statement         { $$ = $1; }
581         | iteration.statement         { $$ = $1; }
582         | jump.statement              { $$ = $1; }
583         ;
584
585 labeled.statement
586         : identifier.or.typedef.name ':' statement { $$ = make_node(';', $3, NULL); }
587         | CASE constant.expression ':' statement  { $$ = make_node(';', $4, NULL); }
588         | DEFAULT ':' statement { $$ = make_node(';', $3, NULL); }
589         ;
590
591 compound.statement
592         : '{' '}'     { $$ = NULL; }
593         | '{' declaration.list '}'  {  $$ = $2; }
594         | '{' statement.list '}'  { $$ = $2; }
595         | '{' declaration.list statement.list '}' {  $$ = make_node(';', $2, $3); }
596         ;
597
598 declaration.list
599         : declaration     { $$ = $1; }
600         | declaration.list declaration { $$ = make_node(';', $1, $2); }
601         ;
602
603 statement.list
604         : statement  { $$ = $1; }
605         | statement.list statement { $$ = make_node(';', $1, $2); }
606         ;
```

```
607
608 expression.statement
609         : expression.opt ';' { $$ = $1; }
610         ;
611
612 selection.statement
613         : IF '(' expression ')' statement                { $$ = make_node(IF, $3, $5); }
614         | IF '(' expression ')' statement ELSE statement { $$ =  make_node('~', $3,  make_node('~', $5, $7 )); }
615         | SWITCH '(' expression ')' statement            { $$ = make_node(IF, $3, $5); }
616         ;
617
618 iteration.statement
619         : WHILE '(' expression ')' statement          { $$ = make_node(IF, $3, $5); }
620         | DO statement WHILE '(' expression ')' ';'   { $$ = make_node(IF, $5, $2); }
621         | FOR '(' expression.opt ';' expression.opt ';'
622                expression.opt ')' statement  { $$ = make_node(IF, $3, $9); }
623         ;
624
625 jump.statement
626         : GOTO identifier.or.typedef.name ';'
627         | CONTINUE ';'                { $$ = make_leaf(void_token); }
628         | BREAK ';'                    { $$ = make_leaf(void_token); }
629         | RETURN expression.opt ';' { $$ = make_leaf(void_token); }
630         ;
631
632
633 /* EXTERNAL DEFINITIONS */
634
635 translation.unit
636         :external.definition { $$ = $1 }
637         | translation.unit external.definition { $$ = make_node('~', $1, $2);}
638         ;
```

```
639
640 external.definition:
641         function.definition
642         | declaration
643         ;
644
645 function.definition
646         :  identifier.declarator compound.statement
647         | declaration.specifier      identifier.declarator compound.statement
648
649         | type.specifier             identifier.declarator compound.statement
650         { $$ = make_node('D', make_node('d', $1, $2), $3); }
651
652
653         | declaration.qualifier.list identifier.declarator compound.statement{
654           $$ = make_node('D', make_node('d', $1, $2), $3); }
655         | type.qualifier.list        identifier.declarator compound.statement {
656           $$ = make_node('D', make_node('d', $1, $2), $3); }
657
658         |                            old.function.declarator compound.statement
659         | declaration.specifier      old.function.declarator compound.statement
660         | type.specifier             old.function.declarator compound.statement
661         | declaration.qualifier.list old.function.declarator compound.statement
662         | type.qualifier.list        old.function.declarator compound.statement
663
664         |                            old.function.declarator declaration.list
665               compound.statement
666         | declaration.specifier      old.function.declarator declaration.list
667               compound.statement
668         | type.specifier             old.function.declarator declaration.list
669               compound.statement
670         | declaration.qualifier.list old.function.declarator declaration.list
```

```
671                 compound.statement
672         | type.qualifier.list        old.function.declarator declaration.list
673                 compound.statement
674         ;
675
676 declarator
677         : typedef.declarator    { $$ = $1; }
678         | identifier.declarator { $$ = $1; }
679         ;
680
681 typedef.declarator:
682         paren.typedef.declarator  /* would be ambiguous as parameter*/
683         | parameter.typedef.declarator   /* not ambiguous as param*/
684         ;
685
686 parameter.typedef.declarator:
687         TYPEDEFname
688         | TYPEDEFname postfixing.abstract.declarator
689         | clean.typedef.declarator
690         ;
691
692     /*  The following have at least one '*'. There is no (redundant)
693      '(' between the '*' and the TYPEDEFname. */
694
695 clean.typedef.declarator:
696         clean.postfix.typedef.declarator   {   }
697         | '*' parameter.typedef.declarator {   }
698         | '*' type.qualifier.list parameter.typedef.declarator   { }
699         ;
700
701 clean.postfix.typedef.declarator:
702         '(' clean.typedef.declarator ')'
```

```
703         | '(' clean.typedef.declarator ')' postfixing.abstract.declarator
704         ;
705
706     /* The following have a redundant '(' placed immediately  to  the
707     left of the TYPEDEFname */
708
709 paren.typedef.declarator:
710         paren.postfix.typedef.declarator
711         | '*' '(' simple.paren.typedef.declarator ')' /* redundant paren */
712         | '*' type.qualifier.list
713                 '(' simple.paren.typedef.declarator ')' /* redundant paren */
714         | '*' paren.typedef.declarator
715         | '*' type.qualifier.list paren.typedef.declarator
716         ;
717
718 paren.postfix.typedef.declarator: /* redundant paren to left of tname*/
719         '(' paren.typedef.declarator ')'
720         | '(' simple.paren.typedef.declarator postfixing.abstract.declarator ')' /* redundant paren */
721         | '(' paren.typedef.declarator ')' postfixing.abstract.declarator
722         ;
723
724 simple.paren.typedef.declarator:
725         TYPEDEFname
726         | '(' simple.paren.typedef.declarator ')'
727         ;
728
729 identifier.declarator
730         : unary.identifier.declarator { $$ = $1; }
731         | paren.identifier.declarator { $$ = $1; }
732         ;
733
734 unary.identifier.declarator
```

```
735         : postfix.identifier.declarator {  $$ = $1; }
736         | '*' identifier.declarator     { $$ =  make_node('*',$2,NULL); } /* $0*/
737         | '*' type.qualifier.list identifier.declarator { }
738         ;
739
740 postfix.identifier.declarator
741         : paren.identifier.declarator postfixing.abstract.declarator  { $$ = make_node('F', $1, $2); }
742         | '(' unary.identifier.declarator ')' {  $$ = $1 }
743         | '(' unary.identifier.declarator ')' postfixing.abstract.declarator {  $$ = make_node('~', $2, $4); }
744         ;
745
746 paren.identifier.declarator
747         : IDENTIFIER  { $$ = make_leaf(lasttok);  }
748         | '(' paren.identifier.declarator ')' { $$ = $1 }
749         ;
750
751 old.function.declarator
752         : postfix.old.function.declarator
753         | '*' old.function.declarator
754         | '*' type.qualifier.list old.function.declarator
755         ;
756
757 postfix.old.function.declarator:
758         paren.identifier.declarator '(' identifier.list ')'
759         | '(' old.function.declarator ')'
760         | '(' old.function.declarator ')' postfixing.abstract.declarator
761         ;
762
763 abstract.declarator
764         : unary.abstract.declarator
765         | postfix.abstract.declarator
766         | postfixing.abstract.declarator
```

```
767         ;
768
769 postfixing.abstract.declarator
770         : array.abstract.declarator { $$ = $1;  }
771         | '(' ')'         { $$ = NULL; }
772         | '(' parameter.type.list ')' {  $$ = $2; }
773         ;
774
775 array.abstract.declarator:
776         '[' ']'  {    $$ = make_node('[',NULL,NULL) ; };
777         | '[' constant.expression ']' {  $$ = make_node('[', $2,NULL) ; }
778         | array.abstract.declarator '[' constant.expression ']' {  $$ = make_node('[',$1, $3) ; }
779         ;
780
781 unary.abstract.declarator
782         : '*' { }
783         | '*' type.qualifier.list  { }
784         | '*' abstract.declarator  { }
785         | '*' type.qualifier.list abstract.declarator { }
786         ;
787
788 postfix.abstract.declarator
789         : '(' unary.abstract.declarator ')'  {  $$ = $2 }
790         | '(' postfix.abstract.declarator ')'     { $$ = $2 }
791         | '(' postfixing.abstract.declarator ')'    { $$ = $2 }
792         | '(' unary.abstract.declarator ')' postfixing.abstract.declarator    { }
793         ;
794
795 %%
```

B.3 Source code for EmamiExtraction.c

```
1  /*  go through each actual param, if it is stored as a func pointer
2   *  in the caller, map it to the actual parameter in the callee
3   *  otherwise it is some other type, so just ignore!
4   *  Only support 1 param per function at the moment..
5   */
6  int emami_map_function_params( NODE* actual_param_tree, CLOSURE* caller, CLOSURE* callee )
7  {
8  #ifdef COMPLEX_CODE
9      return 1;  /* This breaks for very complex code, e.g. variable args and such... */
10 #endif
11
12     char*  mapping_ptr_name   = NULL;
13     char*  mapped_ptr_name    = NULL;
14     EVAR*  mapping_ptr        = NULL;
15     EVAR*  mapped_ptr         = NULL;
16
17     if(actual_param_tree->right != NULL)
18     {
19
20         if(actual_param_tree->right->left->type == 292)
21         {
22             /* get the name of the pointers we want to map from and map too...*/
23             mapping_ptr_name = ((TOKEN*)actual_param_tree->right->left)->lexeme;
24
25             mapping_ptr = emami_hash_lookup_evar(caller->variables,mapping_ptr_name);
26
27             /* Slight hack... Get name of the variable we are mapping from... */
28             mapped_ptr_name = ((TOKEN*)callee->tree->left->right->right->right->left->left)->lexeme;
29
30             /* Get the pointer we are mapping to */
```

```
31                mapped_ptr = emami_hash_lookup_evar(callee->variables,mapped_ptr_name);
32
33               /* Map the pointers of this param to the parameters! */
34               emami_hash_insert_evar(mapped_ptr->pointsto,mapping_ptr);
35          }
36      }
37      return TRUE;
38  }
39
40  /**
41   * Is this function already in the call chain?
42   * If so, then we are going to hit some recursion
43   * return TRUE if recursion exists
44   * otherwise return false
45   **/
46  int emami_check_recursion(  CLOSURE* caller, CLOSURE* callee )
47  {
48     CLOSURE* tmp_closure = NULL;
49
50     tmp_closure = caller;
51
52     while(tmp_closure != NULL)
53     {
54          /* if a previous function and the current one have the same name*/
55          if(strcmp(tmp_closure->lexeme,callee->lexeme) == 0)  return TRUE;
56
57          tmp_closure = tmp_closure->parent;
58     }
59     return FALSE;
60  }
61
62  int emami_print_pointer_call_chain(NODE* tree, CLOSURE* current_function, EVAR* called_pointer, int conditional, int count)
```

```
63 {
64     EVAR* tmp_ptr = NULL;
65     char* func_name = NULL;
66     char* tmp_name = NULL;
67
68     if(called_pointer == NULL) return FALSE;
69
70     int i = 0;
71
72     /* Loop through all the functions that this points to  */
73     while(i < ARR_SIZE)
74     {
75          func_name = called_pointer->functions[i];
76
77         /*  If this function exists! */
78         if(func_name != NULL)
79         {
80             if(count == 0)
81             {
82                 /* So far this is the first & only call so it is definite! */
83                tmp_name = func_name;
84                count = 1;
85             }
86             else
87             {
88                 /* Out  calls are no longer definite */
89                  emami_parse_function_call(tree, func_name, current_function, TRUE);
90                  count = 10;
91             }
92         }
93         i++;
94     }
```

```
95
96      i = 0;
97
98      while(i < HASH_SIZE)
99      {
100         tmp_ptr = called_pointer->pointsto[i];
101
102         while(tmp_ptr != NULL)
103         {
104              count += emami_print_pointer_call_chain(tree, current_function, tmp_ptr, conditional, count);
105              tmp_ptr = tmp_ptr->next;
106         }
107         i++;
108     }
109
110
111     /* Did this pointer contain more than 1 reference? */
112     if(tmp_name != NULL)
113     {
114         if(count == 1)
115         {
116            emami_parse_function_call(tree, tmp_name, current_function, conditional);
117         }
118         else if(count > 1)
119         {
120            emami_parse_function_call(tree, tmp_name, current_function, TRUE);
121         }
122     }
123
124     return count;
125 }
126
```

```
127
128 /**
129  *  Handles a function call
130  *  this can be via a direct
131  *  function call or
132  *  via a call to a pointer..
133  *  simple case is the direct call
134  *
135  *  the various pointer types are more complex
136  **/
137 int emami_parse_function_call(NODE *tree, char* name_func_called, CLOSURE* current_function, int conditional)
138 {
139     CLOSURE* func_called      = (CLOSURE*)get_function_named(name_func_called);
140     CLOSURE* copied_function = NULL;
141     EVAR* func_pointer        = NULL;
142     int i = 0;
143
144     /* If we didnt have a function called that name, then it must be a pointer*/
145     if(func_called == NULL)
146     {
147         func_pointer = emami_hash_lookup_evar(current_function->variables,((TOKEN*)tree->left->left)->lexeme);
148
149         if(func_pointer == NULL)
150         {
151 #ifdef DEBUG
152             printf("[CALL - LIB] \t => \t %s \t\n",((TOKEN*)tree->left->left)->lexeme);
153 #endif
154             emami_push_call(current_function->lexeme,((TOKEN*)tree->left->left)->lexeme, conditional);
155         }
156         else
157         {
158 #ifdef DEBUG
```

```
159             printf("[CALL - PTR] \t => \t %s \t\n",((TOKEN*)tree->left->left)->lexeme);
160 #endif
161
162             emami_print_pointer_call_chain(tree, current_function, func_pointer, conditional, 0);
163         }
164         return TRUE;
165     }
166     /**** direct function call ****/
167
168    /*  ** PROCESS **
169     *
170     *  create a new function based on the template
171     *  This includes all the variables stored in the function
172     *  Sort out the argument list with the correct values
173     *  attach this new function to the previous functions
174     *  list of called functions
175     *  parse the body of the function */
176     copied_function = emami_copy_closure(func_called);
177
178 #ifdef DEBUG
179     printf("[CALL - DIR] \t => \t %s() (%s) \t (EMAMI)\n",copied_function->lexeme,current_function->lexeme);
180 #endif
181
182     emami_push_call(current_function->lexeme, copied_function->lexeme, conditional);
183
184     /* Map the parameters from the calling function to the caller function */
185     emami_map_function_params( tree, current_function, copied_function);
186
187     /* Add the new 'copied_function' to the list of children of this function */
188     emami_add_to_tree( current_function , copied_function );
189
190
```

```
191     /**
192      *  If this node is a recursive
193      *  node, then mark it as such
194      *  and dont dont pass the next node **/
195     if(emami_check_recursion( current_function , func_called ))
196     {
197
198 #ifdef DEBUG
199         printf("[Call - REC] \t => %s \t (EMAMI) \n",func_called->lexeme);
200 #endif
201         copied_function->node_type = RECURSIVE_NODE;
202     }
203     else
204     {
205         /* Begin parsing this new function! Which now has the correct points to set in it... */
206         emami_parse_function_body(copied_function->tree,copied_function,FALSE);
207     }
208     return TRUE;
209 }
210
211 /**
212  *  Do we want to append to this lval or just replace it?
213  *
214  *  When we know that this is DEF. the new value of the variable
215  *  replace it, otherwise we better keep the values that
216  *  may still be needed...
217  *
218  *  Could make this simpler by only keeping false condition in
219  *  but want to make in granular to make it obious or incase
220  *  I change my mind ^_^
221  **/
222 int emami_append(EVAR* lval_evar, int in_conditional)
```

```
223 {
224     if(lval_evar->type == FUNC_ARRAY || lval_evar->type == ARRAY)
225     {
226         /*In an array, so always append*/
227         return TRUE;
228     }
229     else if(lval_evar->type == FUNC_POINTER || lval_evar->type == POINTER)
230     {
231         if(in_conditional == TRUE)
232         {
233             /* Inside a conditonal statement, append!*/
234             return TRUE;
235         }
236         else
237         {
238             /* not in a conditional, replace linkage! */
239             return FALSE;
240         }
241     }
242     /* We are not certain, err on side of caution */
243     return TRUE;
244 }
245
246 /**
247  *  This function is called when an assignmnet is made
248  *  checks are performed to see
249  *  a) What type of object is being assigned to
250  *  b) what are we assigning
251  *
252  *  the correct assignment procedure is performed based
253  *  on these two variables
254  **/
```

```
255 int emami_perform_assignment(char* lval_lexeme,
256                              char* rval_lexeme,
257                              int in_conditional,
258                             EVAR** emami_vars)
259 {
260     EVAR* lval_evar = emami_hash_lookup_evar(emami_vars,lval_lexeme);
261     EVAR* rval_evar= emami_hash_lookup_evar(emami_vars,rval_lexeme);
262 
263     if(lval_evar == NULL) return FALSE; /* You cant assign something to nothing... */
264 
265     /* Yes this is big and disgusting, but can't think of a better way of doing all conditionals...
266      * If the rval returns NULL, we know it is not a pointer, hence actual function*/
267      if(rval_evar == NULL)
268     {
269         if(emami_append(lval_evar, in_conditional) == TRUE)
270         {
271             emami_add_function(lval_evar,rval_lexeme);
272             return TRUE;
273         }
274         else
275         {
276             emami_replace_function(lval_evar,rval_lexeme);
277             return TRUE;
278         }
279     }
280     else
281     {
282         if(emami_append(lval_evar,in_conditional) == TRUE)
283         {
284             emami_add_pointsto(lval_evar,rval_evar);
285             return TRUE;
286         }
```

```
287         else
288         {
289             emami_replace_pointsto(lval_evar,rval_evar);
290             return TRUE;
291         }
292     }
293     return TRUE;
294 }
295
296 int emami_parse_function_body(NODE *tree, CLOSURE* current_function, int in_conditional)
297 {
298      if(tree == NULL || current_function == NULL)
299     {
300         return FALSE;
301     }
302
303     switch(tree->type)
304     {
305         case (IF) :
306         {
307            if(tree->type==LEAF)
308            {
309                  // printf("leaf...\n");
310                  return;
311            }
312
313            if(tree->left != NULL)
314            {
315                emami_parse_function_body(tree->left,current_function, FALSE);
316            }
317
318            if(tree->right != NULL)
```

```
319             {
320                 emami_parse_function_body(tree->right,current_function, TRUE);
321             }
322 
323             break;
324         }
325         case (int)'=' :
326         {
327             if(tree->right->type == (int)'&')
328             {
329                 emami_perform_assignment(((TOKEN*)tree->left->left)->lexeme,
330                                          ((TOKEN*)tree->right->left->left)->lexeme,
331                                          in_conditional,
332                                          current_function->variables);
333             }
334             else
335             {
336                  emami_perform_assignment(((TOKEN*)tree->left->left)->lexeme,
337                                          ((TOKEN*)tree->right->left)->lexeme,
338                                          in_conditional,
339                                          current_function->variables);
340             }
341             break;
342         }
343 
344         case APPLY :
345         {
346              emami_parse_function_call(tree, ((TOKEN*)tree->left->left)->lexeme,current_function,in_conditional);
347              /* HACK: let this slip into the default case
348               * as we want to see if there are any nested function calls
349               * in this call! <- May need to investiagte, dont
350               * think this will work for certain cases, e.g.
```

```
351                 *  f(a(),b());  */
352                 return TRUE;
353             }
354             default :
355             {
356                 if(tree->type==LEAF)
357                 {
358                       return;
359                 }
360
361                 if(tree->left != NULL)
362                 {
363                     emami_parse_function_body(tree->left,current_function,in_conditional);
364                 }
365
366                 if(tree->right != NULL)
367                 {
368                      emami_parse_function_body(tree->right,current_function,in_conditional);
369                 }
370             }
371         }
372         return TRUE;
373 }
374
375
376 int emami_create_variables(NODE *tree, EVAR** emami_vars)
377 {
378         if(tree == NULL)
379         {
380             return FALSE;
381         }
382
```

```
383     switch(tree->type)
384     {
385         case 126 :  /* Tilde */
386         {
387             if(tree->right->type == (int)'*')
388             {
389                 if(tree->right->left->type == 126) /* Array declaration */
390                 {
391                     emami_push_var(((TOKEN*)tree->right->left->left->left->left)->lexeme,emami_vars,ARRAY);
392                  }
393                  else
394                  {
395                     emami_push_var(((TOKEN*)tree->right->left->left)->lexeme,emami_vars,POINTER);
396                  }
397             }
398             else if(tree->right->type == 126) /* Tilde - Function pointers */
399             {
400                  if(tree->right->left->left->type == (int)'F') /* Array declaration */
401                  {
402                     emami_push_var(((TOKEN*)tree->right->left->left->left->left)->lexeme,emami_vars,FUNC_ARRAY);
403                  }
404                  else
405                  {
406                     emami_push_var(((TOKEN*)tree->right->left->left->left)->lexeme,emami_vars,FUNC_POINTER);
407                  }
408              }
409              else
410              {
411              }
412         return TRUE;
413     }
414     default :
```

```
415     {
416         if(tree->type==LEAF) return TRUE;
417         if(tree->left != NULL) emami_create_variables(tree->left,emami_vars);
418         if(tree->right != NULL) emami_create_variables(tree->right,emami_vars);
419     }
420   }
421  return TRUE;
422 }
```

B.4 Source code for ExtractionInterface.c

```
1  int naive_extraction(NODE* tree, char* name)
2  {
3       CLOSURE* tmp_closure = NULL;
4
5       graphsee_clean_program(name, NAIVE);
6       graphsee_add_program(name, NAIVE);
7
8       extract_top_functions(tree);
9       tmp_closure = head_closure;
10
11      while(tmp_closure != NULL)
12      {
13           graphsee_add_function( name , tmp_closure->lexeme , NAIVE);
14
15           naive_parse_generate_funcs(tmp_closure->tree, name);
16           naive_parse_generate_calls(tmp_closure->tree, name, tmp_closure->lexeme , 0);
17
18           tmp_closure = tmp_closure->next;
19      }
20
21      graphsee_print_function_buffer();
22      graphsee_print_call_buffer();
23
24     return TRUE;
25 }
26
27 int emami_extraction(NODE* tree, char* name)
28 {
29     CLOSURE* tmp_closure = NULL;
30     CLOSURE* current_closure = NULL;
```

```
31
32      UNODE**       nodes_arr;
33
34      extract_top_functions(tree);
35
36      /* Define global vars for all functions */
37       tmp_closure = head_closure;
38
39       while(tmp_closure != NULL)
40       {
41  #ifdef DEBUG
42           printf("[CREATE] \t => \t %s\n",tmp_closure->lexeme);
43  #endif
44            emami_create_variables(tmp_closure->tree,tmp_closure->variables);
45            tmp_closure = tmp_closure->next;
46       }
47
48      current_closure = get_function_named("main");
49
50      if(current_closure != NULL)
51      {
52  #ifdef DEBUG
53           printf("Found a main - begin interpreting\n");
54  #endif
55           emami_parse_function_body(current_closure->tree, current_closure, FALSE);
56      }
57      else
58      {
59           printf("Emami Algorithm Requires Main to parse code\n");
60           exit(1);
61      }
62      /* ... */
```

```
63 }
64
65
66 int steens_extraction(NODE* tree, char* program_name)
67 {
68     CLOSURE* tmp_closure = NULL;
69     UNODE**       nodes_arr;
70
71      graphsee_clean_program(program_name, STEENS);
72      graphsee_add_program(program_name, STEENS);
73
74
75     extract_top_functions(tree);
76     tmp_closure = head_closure;
77
78     while(tmp_closure != NULL)
79     {
80         graphsee_add_function( program_name , tmp_closure->lexeme , STEENS);
81
82         nodes_arr = uf_create_unodes();
83
84         steens_create_variables(tmp_closure->tree->right,nodes_arr);
85         steens_union_variables(tmp_closure->tree->right,tmp_closure->lexeme, program_name, nodes_arr);
86
87         uf_extract_function_list(nodes_arr, tmp_closure->lexeme, program_name);
88
89         tmp_closure = tmp_closure->next;
90     }
91
92
93     graphsee_print_function_buffer();
94     graphsee_print_call_buffer();
```

```
95  }
96
97
98  /**
99   * In this stage we create a list of all
100  * the variables defined in the
101  * function that is being parsed
102  **/
103 int steens_create_variables(NODE *tree, UNODE** node_arr)
104 {
105     if(tree == NULL)  return FALSE;
106
107     switch(tree->type)
108     {
109         case 126 :  /* Tilde */
110         {
111             if(tree->right->type == (int)'*')
112             {
113                 if(tree->right->left->type == 126) /* Array declaration */
114                 {
115                     uf_push_unode(((TOKEN*)tree->right->left->left->left->left)->lexeme,node_arr);
116                 }
117                 else
118                 {
119                     uf_push_unode(((TOKEN*)tree->right->left->left)->lexeme,node_arr);
120                 }
121             }
122             else if(tree->right->type == 126) /* Tilde - Function pointers */
123             {
124                 if(tree->right->left->left->type == (int)'F') /* Array declaration */
125                 {
126                     uf_push_unode(((TOKEN*)tree->right->left->left->left->left)->lexeme,node_arr);
```

```
127                     }
128                     else
129                     {
130                         uf_push_unode(((TOKEN*)tree->right->left->left->left)->lexeme,node_arr);
131                     }
132                 }
133                 else
134                 {
135                 }
136             return TRUE;
137         }
138         default :
139         {
140             if(tree->type==LEAF)  return TRUE;
141             if(tree->left != NULL) steens_create_variables(tree->left,node_arr);
142             if(tree->right != NULL)  steens_create_variables(tree->right,node_arr);
143         }
144       }
145     return TRUE;
146 }
147
148
149 int naive_parse_generate_funcs(NODE *tree, char* program_name)
150 {
151     if(tree == NULL) return FALSE;
152
153     switch(tree->type)
154     {
155         case APPLY:
156         {
157            graphsee_add_function( program_name , ((TOKEN*)tree->left->left)->lexeme , NAIVE);
158         }
```

```
159         default :
160         {
161             if(tree->type==LEAF) return TRUE;
162             if(tree->left != NULL) naive_parse_generate_funcs(tree->left,program_name);
163             if(tree->right != NULL) naive_parse_generate_funcs(tree->right,program_name);
164         }
165     }
166     return TRUE;
167 }
168
169 int naive_parse_generate_calls(NODE *tree, char* program_name, char* caller, int i)
170 {
171     if(tree == NULL) return FALSE;
172
173     switch(tree->type)
174     {
175         case APPLY:
176         {
177             graphsee_add_call( program_name, caller , (((TOKEN*)tree->left->left)->lexeme) , i , FALSE, NAIVE);
178             i++;
179         }
180
181         default :
182         {
183             if(tree->type==LEAF) return 0;
184
185             if(tree->left != NULL)
186             {
187                 i += naive_parse_generate_calls(tree->left, program_name, caller,i);
188             }
189
190             if(tree->right != NULL)
```

```
191             {
192                 i += naive_parse_generate_calls(tree->right, program_name, caller,i);
193             }
194         }
195     }
196     return i;
197 }
```

B.5 Source code for UnionFind.c

```
1  /**
2   *  Loop through all the point-to set
3   *  in nodes_arr and extract the functions
4   *  that had a chance of being called
5   *  via function pointer calls
6   **/
7  char** uf_extract_function_list(UNODE** nodes_arr, char* caller, char* program_name)
8  {
9      int nodes_index = 0;
10     int tmp_funcs_index = 0;
11     int current_index = uf_get_size(nodes_arr);
12     char ** funcs;
13
14     while(nodes_index < current_index)
15     {
16 #ifdef DEBUG
17         /*  Prints the atoms in the set by their order  */
18         uf_printset(nodes_arr[nodes_index]);
19 #endif
20         /* get the functions called from this set,
21          * but only if it is a leaf
22          * (i.e.) has no lower nodes*/
23         if(nodes_arr[nodes_index]->leaf == TRUE)
24         {
25             funcs = uf_functions_called_in_set(nodes_arr[nodes_index]);
26         }
27         else
28         {
29             funcs = NULL;
30         }
```

```
31
32          /*  Simple print code...  */
33          if(funcs != NULL)
34          {
35              while(funcs[tmp_funcs_index] != NULL)
36              {
37                  graphsee_add_function( program_name , funcs[tmp_funcs_index] , STEENS) ;
38                  graphsee_add_call( program_name,  caller , funcs[tmp_funcs_index] , 0 , FALSE, STEENS);
39
40                  tmp_funcs_index++;
41              }
42                tmp_funcs_index = 0;
43           }
44          nodes_index++;
45      }
46
47      return funcs;
48  }
49
50  /**
51   *  Extracts all the funcions that this set
52   *  (starting with UNODE node)
53   *  calls with the following provisos:
54   *
55   *  If no pointer is called ptr();
56   *  in the set, the empty set { }
57   *  is returned
58   *
59   *  Otherwise a list of function names
60   *  are returned this is a union of
61   *  all the sets of functions that were
62   *  assigned to the various poitners
```

```
63  **/
64 char** uf_functions_called_in_set(UNODE* node)
65 {
66      /* all the functions called in this list */
67      UNODE* tmp_node                 = NULL;
68      char** functions_called         = (char**)calloc(100,sizeof(char*));
69      char*  tmp_func                 = NULL;
70      int    node_functions_index     = 0;
71      int    current_func_list_index  = 0;
72      int    func_arraIX              = 0;
73      int    found_same               = FALSE;
74
75       /* was any of the pointers called as a function... */
76      int called = FALSE;
77
78      tmp_node = node;
79
80      while(tmp_node != NULL)
81      {
82         /* Was one of the pointers called? */
83         if(tmp_node->called == TRUE)
84         {
85               called = TRUE;
86         }
87
88          /* Going to check through each function
89           * assigned to this pointer
90           * and add it to list of possible called functions */
91          tmp_func = tmp_node->functions[0];
92          node_functions_index = 0;
93          current_func_list_index = 0;
94
```

```
95          while(tmp_node->functions[node_functions_index] !=  NULL)
96          {
97              while(functions_called[current_func_list_index] != NULL)
98              {
99                 /* If we found the same function, we dont want to add it to list */
100                 if(strcmp(functions_called[current_func_list_index],tmp_func) == 0)
101                 {
102                     found_same = TRUE;
103                 }
104                 current_func_list_index++;
105             }
106
107             /* Add it to list if it has not been found */
108             if(found_same != TRUE)
109             {
110                 functions_called[func_arraIX++] = tmp_func;
111             }
112
113             found_same = FALSE;
114             tmp_func = tmp_node->functions[++node_functions_index];
115          }
116
117             /* Let's look at the next node in this set */
118             tmp_node = tmp_node->parent;
119      }
120
121      /* If one of the pointers was called, then *all* of them could have be called */
122      if(called == TRUE)
123      {
124         return functions_called;
125      }
126      return NULL;
```

```
127 }
128
129 /**
130  *  Union two nodes from nodes_arr
131  *  so that they form 1 set.
132  *
133  *  process: Find root node for each node
134  *  (each node may already part of a bigger set)
135  *
136  *  check to see if the roots are the same, if they are
137  *  the nodes are already union'd! (return)
138  *
139  *  otherwise, union the sets such that the left node is the parent
140  *  of the right node and that the left node is no longer set as aleaf
141  *  (note: this does not have any effect on the leaf status of the
142  *   other node)
143  **/
144 int uf_union_unodes(char* left_lexeme,char* right_lexeme, UNODE** nodes_arr)
145 {
146     UNODE* left_node = uf_find_root_unode(left_lexeme,nodes_arr);
147     UNODE* right_node = uf_find_root_unode(right_lexeme,nodes_arr);
148
149     if(left_node == NULL || right_node == NULL)
150     {
151         return FALSE;
152     }
153
154     /* are they part of the same set? */
155     if(left_node != right_node)
156     {
157          right_node->parent = left_node;
158          left_node->leaf    = FALSE;
```

```
159 #ifdef DEBUG
160     printf("UNION( %s ,%s )\n",left_node->lexeme,right_node->lexeme);
161 #endif
162     }
163     return TRUE;
164 }
165
166 /**
167  * Function pushes a map between a
168  * unode and a function
169  */
170 int uf_push_function_map(char* node_lexeme, char* func_lexeme, UNODE** nodes_arr)
171 {
172     int i = 0;
173
174      /* grab the node we want to append to... */
175     UNODE* tmpNode = uf_find_unode(node_lexeme,nodes_arr);
176
177     if(tmpNode == NULL)
178     {
179         return FALSE;
180     }
181 #ifdef DEBUG
182     printf("[MAP] \t => \t %s (node) : %s (function)\n",node_lexeme,func_lexeme);
183 #endif
184     /* Find the end of the current function list */
185     while(tmpNode->functions[i] != NULL) i++;
186
187     tmpNode->functions[i] = func_lexeme;
188
189     return TRUE;
190 }
```